VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141

valFORTH T.M.
SOFTWARE SYSTEM
for ATARI*

valFORTH 1.1 T.M.

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

valFORTH™
SOFTWARE SYSTEM

valFORTH™ 1.1

Stephen Maguire
Evan Rosen

(Atari interfaces based on work by Patrick Mullarky)

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

valFORTH 1.1 USER'S MANUAL

Table of Contents

STROLLING THROUGH valFORTH 1.1

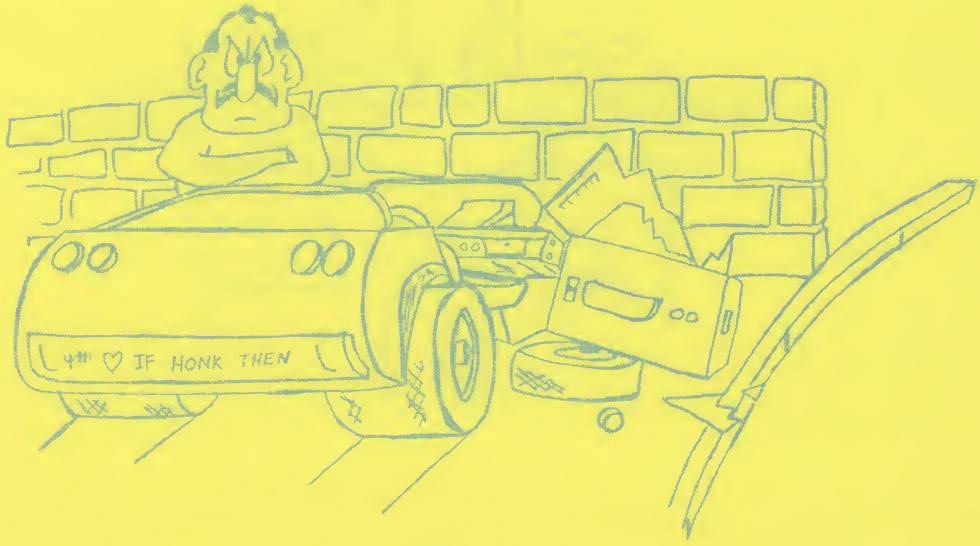
Welcome. For this excursion you'll need an ATARI 800 (or 400) with at least 24K, a disk drive, monitor, a printer, and valFORTH 1.1. You could even do without the printer. Please get everything up and running, and boot valFORTH.

(To boot the disk, turn the drive(s) on and the computer off. Insert the disk in drive 1 and turn the computer on. The disk should now be booting, and the monitor speaker should be going beep-beep-beep-beep as valFORTH loads.)

ERRORS, RECOVERIES, CRASHES

Before we get started, let's mention the inevitable: Most of the time when you make an error you'll receive one of the fairly lucid fig-Forth error messages. If you just get a number, this will probably refer to the Atari error message list which you can find in the documentation that came with your computer. Since the Atari is a rather complex beast, you may sometimes get into a tangle that looks worse than it is. Keep your head. If you have party-color trash on the screen, for instance, and yet you can still hear the "peek-peek-peek" of the key when you hit return, you may have merely blown the display list without hurting your system. Try Shift-Clear followed by 0 GR. . Very often you're home again. If this doesn't work, try a warm start: Hold down a CONSOLE button, say START, and while you've got it down, press SYSTEM RESET and hold both for a moment until the "valFORTH" title comes up. (If you were to push the SYSTEM RESET button alone, you'd get a cold start, which takes you back to just the protected dictionary.) A warm start gets you back to the "ok" prompt without forgetting your dictionary additions. If warm start doesn't work, your system is being kept alive only by those wires connected to it; it no longer has a life of its own. The standard procedure now is to push SYSTEM RESET alone a few times (cold start) in a superstitious manner, and then reboot the system.

Look carefully at the code that blew the system last time. If you're really having trouble debugging, sprinkle a bunch of WAIT's and/or .S's (Stack Printouts) through the code, and go through again. The best thing about those first few long debugging sessions in any computer language is that they teach you the value of writing code carefully.



FORMATTING AND COPYING DISKS

You may have noticed that your system came up in a green screen. In a little while you'll be able to change it to anything you like. We'll get to that in a moment, but right now type 170 LIST (and then hit RETURN.) Behold the table of contents. Our first priority should be to make a working disk by copying the original.

Let's assume that you have a blank, unformatted disk on which to make your copy. Notice the line called FORMATTER on screen 170. At the right side of this line is probably 92 LOAD, though the number may be different in later releases. Type 92 LOAD (or whatever the number is) and wait until the machine comes back with "ok". Now you're going to type FORMAT, but for safety's sake why not remove the valFORTH disk and insert the blank disk? One never knows if newly purchased software will give you warnings before taking action. ("Warnings" or "Prompts" make a system more friendly.) Ok, now type FORMAT. For the drive number you probably want to hit "1", unless you've got more than one drive and don't want to format on the lowest. In answer to the next prompt, hit RETURN unless you've changed your mind. Now wait while the machine does the job. If you get back "Format OK" you're in business. (If "Format Error" comes back, suspect a bad blank disk or drive.) You might as well format another disk at this time on which to store your programs.

Now to make the copy. Return the valFORTH disk to the drive and do 170 LIST again. Find DISK COPIERS and do 72 LOAD, or whatever number is indicated. When the "ok" prompt comes back, two different disk copying routines are loaded: DISKCOPY1 for single drive systems and DISKCOPY2 for multiple drive systems. Type whichever of these words is appropriate and follow the instructions. ("source" means the disk you want to copy. "dest." is the blank "destination" disk.) There are 720 sectors that have to be copied. Since this can't be done in one pass, if you are using DISKCOPY1 you will have to swap the disks back and forth until you're done. (The computer will tell you when.) The less memory you have, the more passes; there is great benefit in having 48K. If you have more than one drive, it still takes several internal passes, but there is no swapping required. Either way, the process takes several minutes with standard Atari disk drives.

Nice going. Now store the original disk in some safe place. Don't write protect your copy yet. First we'll adjust the screen color to your taste. Just to see if you really have a good copy, boot it. This can be done by the usual on-off method, or by typing BOOT.

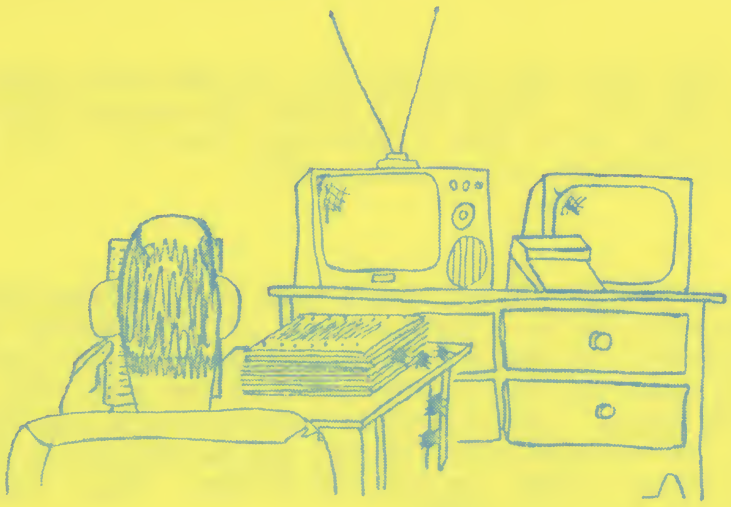
COLORS

Before playing with the colors, let's look at something else. Type VLIST, and watch the words go by. These are all of the commands that are currently in the "dictionary" in memory in your system. You can cause this listing, or any other, to pause by hitting CTRL and 1 at the same time. This is a handy feature of the Atari. The listing is restarted with the next CTRL 1. Additionally, in valFORTH most listings may be aborted by pressing any of the three yellow buttons START, SELECT, and OPTION. These three buttons together will be referred to as the CONSOLE.

Do VLIST again, and abort it with a CONSOLE press after a few lines. At the top of the list you should see the word TASK. Remember that for a moment. Do 170 LIST again. Look over the list and find COLOR COMMANDS, and LOAD as appropriate. Now do VLIST again, and stop the list when it takes up about half of the screen. Above TASK you now can see a number of new commands, or "words" as they are commonly called in Forth. These were added to the dictionary by the LOAD command. Here's what some of these words do:

Type BLUE 12 BOOTCOLOR, and you get a new display color. Try BLUE 2 BOOTCOLOR. If you try this action with the number as 4, the letters will disappear, and you'll have to type carefully to get them back. The number is the luminance or "lum" and is an even number in the range 0 to 14. The color-name is called the "hue." The word "color" will be used to refer to a particular combination of hue and lum; hence PINK 6 is a color, PINK is a hue. There are 16 hues available and you can read their names from the display, starting with LTORNG ("light orange") and ending with GREY. (The hues may not match their names on your monitor. Later on you'll be able to change the names to your liking, or eliminate them altogether to save memory, and just use numbers. For instance, PINK is equal to 4.)

Try out different colors using BOOTCOLOR, until you find one you can live with for a while. We usually use GREEN 10 or GREEN 12 in-house at Valpar. While you are doing this you'll probably make at least one mistake, and the machine will reply with an error message like "stack empty." Just hit return to get the "ok" back and start whatever you were doing again. Actually, you don't even have to get the "ok" back, but it's reassuring to see it there. When you've got a color you like, do VLIST again. Note the first word above TASK. It should be GREY. Carefully type FORGET GREY, and do VLIST again. Notice that GREY and all words above it are indeed forgotten. That's just what we want. Now type SAVE. You'll get a (Y/N) prompt back to give you a chance to change your mind, since SAVE involves a significant amount of writing to drive #1. For practice, check to see that you still have the copy in drive one, and if it is there, hit Y, and off we go. When "ok" comes back, remove the disk and apply a write protect tab to it. Boot this disk again to see that it will come up in your selected color.



DEBUGGING

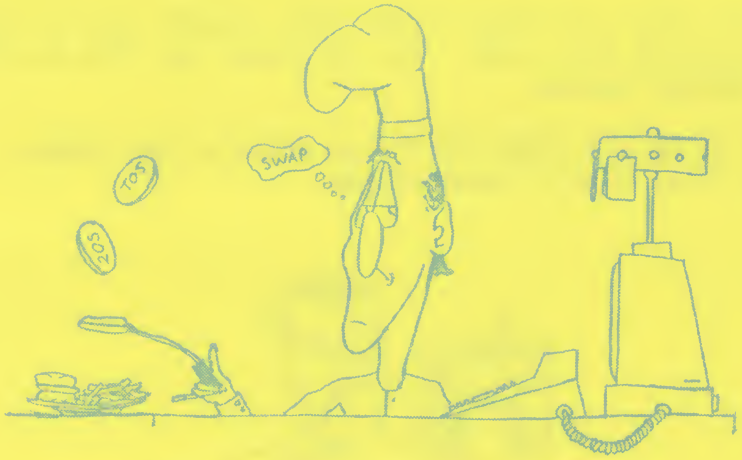
Look at 170 again. Load the DEBUGGING AIDS, and type ON STACK. You'll see that the stack is empty. Great, what's a stack? The best answer to this is to suggest that you read Leo Brodie's book, Starting FORTH. This amusing and thorough treatment of FORTH starts from the novice level and continues on to most of the advanced concepts in FORTH. Starting FORTH is available from many sources, including Valpar International. Included with your valFORTH package is a document called "Notes on Starting FORTH for the fig-FORTH User" which pinpoints differences between Brodie's dialect of FORTH, called 79 Standard, and the somewhat more common fig-FORTH on which valFORTH was based.
It is not feasible to present a course on FORTH in these pages, since FORTH is far more powerful than BASIC, which itself requires a fair amount of space to present. However, we'll try to be as considerate as possible to the FORTH-innocent.

The visible stack is a very good debugging and practice tool. Type a few integers, say 5 324 -19 0 and hit RETURN. The numbers are now visible on the stack. Top of stack, TOS, is at right. Print the top entry by typing " . " and then do DUP. Note that there are now two -19's on top. Do " * " to multiply them together. Now do DROP to discard the product, 361, currently at TOS. Ok, now do SWAP to exchange the 324 and 5 and then do " / " to divide 324 by 5. This should leave 64, since the answer is truncated. Now type 1000 * and notice that instead of getting 64000 you get -1536. This is of course two's complement on a two-byte number. Type U.S which will switch the visible stack to unsigned representation. Type .S to go back. The words .S and U.S may be used with the visible stack on or off to show the stack one time. Now type OFF STACK. Type in a few more numbers, say 1 2 3 4 5 6 7. ON STACK again, and observe that the entries are retained. Do OVER to bring a copy of the 6 over the 7. Now DROP it. Do ROT to rotate the third from top, 5, to the top. Now do <ROT to put it back. In addition to all of these normal routines, valFORTH supports PICK and ROLL both coded in 6502 for speed. Notice that the 5th on stack is a 3. 5 PICK will bring a copy of it to TOS. Do this and then DROP the 3. Do 5 ROLL to pull the 3 out of the stack and place it at TOS. DO SP! to clear the stack.

One point about number bases: Right now you're in DECIMAL. By typing HEX you go into hexadecimal, and typing DECIMAL or its abbreviation DCX you get back. And, as usual, virtually any base may be used by typing N BASE ! where N is the base you want. Thus, 2 BASE ! gives binary, etc. Some errors, particularly during loading, may leave you in an unexpected base, like base 0, for instance. If you find the machine acting normally except for numbers, this may have happened. A simple DCX will get you back to decimal. The word B? will print the current base in decimal. Put 30 on the stack and then do HEX. Now do B?. Do DCX to return to decimal.

While we're on the subject of numbers, do ON and note that it is just a CONSTANT equal to 1. Similarly, do OFF and see that it is zero. Try 0 STACK and then 1 STACK. The words ON and OFF are provided to enhance readability of code, but could be substituted by 1 and 0 if desired. The two representations are equally fast.

We mention to the newcomer to FORTH that the stack takes the place of dummy variables or dummy parameters in other languages. This reduces memory overhead in several ways but does exact a penalty of reduced readability of FORTH source code. Consistent and sensible source code formatting can significantly enhance readability. The source code on the present disk may be used as a reasonably good example of well-arranged code.



Now a few words about DECOMP. Clear the stack. Type in 3 and 4. Do OVER OVER followed by 2DUP and notice that these two phrases have the same effect. Clear the stack and then turn it off if you like, and do DECOMP 2DUP. What you see is a decompilation of 2DUP which indicates that it is indeed defined as OVER OVER. Decomp OVER. The word "primitive" in the decompilation of OVER indicates that OVER is defined in machine code.

Decomp LITERAL. The word (IMMEDIATE) after LITERAL in the decompilation indicates that LITERAL is immediate. Not all words can be decompiled by DECOMP, and sometimes trash will be printed with long pauses between lines. In this case, hold down any CONSOLE button (the three yellow ones, remember) until the "ok" comes back. This may take several seconds, but rarely much longer.

PRINTING

If you have a printer attached, we can generate some hardcopy. Look at screen 170 again. You can see the line labeled PRINTER UTILITIES. Don't load it, though. The printer utilities were loaded automatically when you loaded the debugging aids, and so are in the dictionary already. (There is no need to have them in twice, though it wouldn't hurt.) You have access to the words P:, S:, LISTS, PLISTS, PLIST, and a couple of others relating to output. Do VLIST and see if you can spot this group. As a matter of fact, do ON P: VLIST OFF P: all in one shot. ON P: is used to route output to the printer or not. OFF P: stops sending to the printer. Try ON P: OFF S: 170 LIST CR OFF P: ON S: and notice that this time text is not sent to the display screen, only to the printer. That's because of OFF S: .

Look at screen 170 again, either on display or in hardcopy, and note which screen the printer utilities start. Type this number in, but don't type load. Instead, after the number, type 10 PLISTS. This prints 10 screens starting from the first screen you just typed in. If you have a reasonably smart printer, it will automatically paginate, so that the screens are printed three to a page. If the printer acts peculiarly after printing each third screen, the pagination code in the word EJECT is probably not right for your printer. You'll be able to change this later on.

Now type 30 150 LISTS and after a few blank screens you'll see the entire disk go by, except for the boot code. You can pause any time by CTRL 1 or stop by holding a CONSOLE button.

Finally, do ON P: 30 179 INDEX OFF P: to print a disk index. The index is made up of the first line of each screen.

EDITING

Two editors have been included in this package. The fig (Forth Interest Group) Editor and the valFORTH 1.0 Editor. The latter, while a perfectly useable video-display editor in its own right, is actually a stripped-down version of the valFORTH 1.1 Editor, available with the Utilities/Editor package from Valpar International. The 1.0 Editor is provided to give the user some idea of what the very powerful 1.1 Editor is like, without actually providing it. (Among other things, the 1.1 Editor has a user-definable line buffer of up to 320 lines with a 5 line visible window at the bottom of the display. This window can be seen at the bottom of the 1.0 Editor, but is inactive.)

The fig Editor is a general-purpose FORTH line editor, and was the FORTH editing workhorse until good video-displays were developed.

The fig Editor User Manual is located just after this section. It is based on that by Bill Stoddart of FIG, United Kingdom, published in the fig-Forth installation manual 10/80, and is provided through the courtesy of the FORTH INTEREST GROUP, P.O. Box 1105, San Carlos, CA 94070. Serious Forth programmers should write FIG to request their catalog sheet of references and publications.

Let's look at the valFORTH editor 1.0. Refer to the directory again, screen 170, and load the valFORTH editor. (Don't load the fig Editor by mistake.) Before proceeding, make sure that the write-protect tab on your disk is secure. The word to enter the editor at the screen on top of stack is V. You can remember it by thinking of it as "view." Type 170 V. Screen 170 is now on the display again, but in the valFORTH 1.0 Editor rather than as a listing. This Editor is a subset of the valFORTH 1.1 Editor available in the Editor/Utilities package, which is MUCH more powerful and convenient, and is priced far lower than any comparable product of which we are aware. The Editor Command card provided shows all of the commands available with the 1.1 Editor. Commands available with the 1.0 Editor are marked with asterisks (*) on the card. Let's run through them:

The cursor can be moved as in the Atari "MEMO PAD" mode. That is, hold down the control key (CTRL) and move the cursor around the display with the four arrow keys. To enter text (replace mode only in 1.0), position the cursor and type it in. Delete characters with the backspace key as usual. The cursor will wrap to the next line at the end of a line, and to the top of the screen when it goes off the bottom. You can type at will on this screen since we won't save the changes to disk.

Do a Shift-Insert and notice that a blank line is inserted at the cursor line. The bottom line is lost, though it is recoverable in the 1.1 version. Now do Shift Delete to remove a line. (Delete is on the Backspace key). These are all of the Editing commands available in the 1.0 Editor. There are two methods of exiting the editor, CTRL S and CTRL Q. CTRL S marks the screen for saving to disk, and CTRL Q forgets the latest set of editing changes. As usual, changes are not saved immediately. This is accomplished with the word FLUSH or by bringing other screens into the buffers and pushing the edited ones out. Again, as usual, the EMPTY-BUFFERS command, or its valFORTH abbreviation, MTB, will clear all buffers, thus forgetting any changes that have not yet been written to disk.

Try CTRL Q to exit now. Reedit the screen by typing L. L does not require an argument on stack and will bring the last-editing screen into the editor. The words CLEAR and COPY have their normal meanings, as does WHERE, which has had the standard fig bug fixed. See the glossary for details. Note that since COPY in valFORTH does not FLUSH its changes, careful use allows transfers of single screens between disks by swapping disks after COPY and before FLUSH. This is particularly handy, for example, for transferring error message screens 176-179 between disks.

You can make this transfer by doing

```
176 176 COPY 177 177 COPY 178 178 COPY 179 179 COPY
```

and then swapping in the destination disk and typing FLUSH. You may want to define a word to do this automatically:

```
: ERRXFR                          ( -- )
  CR   ." Insert source and press START" WAIT
  180 176
  DO I I COPY
  LOOP
  CR ." Insert dest. and press START" WAIT
  FLUSH ;
```

Because there are four 512 character screen buffers in memory in valFORTH, four 512 characters screens at a time is the maximum for this method. Bulk screen moves on a single disk or between disks are available with the Utilities/Editor Package.

Note: The word "screen" in Forth refers to an area of the disk. When you do 170 LIST you are listing screen 170. In valFORTH there are 180 screens, numbered 0-179, on the disk in drive 1. In multiple-drive systems screen numbers continue across drives, so that screens 180-349 are on drive 2. 180 LIST will automatically read from drive 2. For technical reasons screen 0 should not be used for program code.

Whichever editor you use for the moment, you can write your programs to a blank disk and load them from there. Remember that in fig-FORTH (and so also in valFORTH), if you wish to continue loading from one screen to the next, all but the last screen should end in -->. You'll see this all through the valFORTH 1.1 code. You'll also see ==>. For present purposes you can use --> everywhere, and forget about ==>. ==> is actually a "smart" version of --> that does nothing if the system uses 1024 character screens instead of 512.

If you are a FORTHER, and wish to use 1024 byte screens, do FULLK. To return to 512 character screens, do HALFK. (A working disk may be SAVE'd in either condition.) Note that the valFORTH 1.0 Editor will not edit 1024 character screens, though the 1.1 version will, and includes special 1K notation. In the same vein, the word KLOAD that appears in the source code is a smart load. See the Glossary for details.

To terminate loading one simply omits the --> on the last screen. ;S may be used to end loading at any point. Also note that valFORTH --> and ==> are smart in the sense that if you wish to stop loading before the machine is ready to stop, simply hold down a CONSOLE button. When --> or ==> execute, they first check the CONSOLE. If a button is pressed, they stop loading instead of continuing with the next screen.

Before leaving editing practice, type MTB to empty the disk buffers and assure yourself that nothing will be flushed to disk accidentally as you read in new screens. Or else, do FLUSH if you really want to save your changes. (Remember to remove the write-protect tab if you do.)

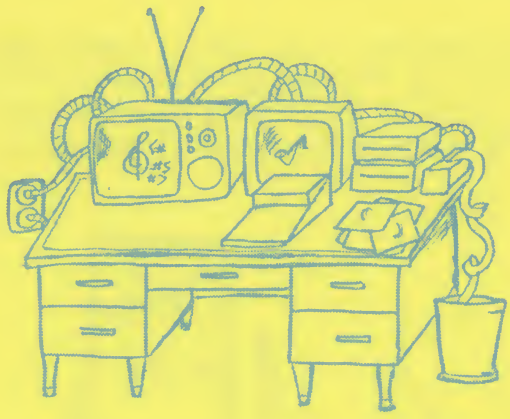
GRAPHICS

On to Graphics. Check screen 170 and load the Color Commands again, and then the Graphics Package. VLIST to see what you've got, and print the list if you like. You may notice that GR. is not among these freshly loaded words: It is in the kernel, that is, the booted code. Try the following sequence:

```
2 GR. (BASIC Graphics mode 2)
5 5 POS. (Move the graphics cursor)
G" TEST" (Send text to graphics area)
1  COLOR (Pick a new graphics color)
G" TEST" (More text)
: SMPL  4 0 DO I  COLOR G" TEST" LOOP ; (automate)
SMPL SMPL SMPL (Try it out)
: MANY BEGIN SMPL ?TERMINAL UNTIL ; (More automation)
1 GR. (Go somewhere else)
MANY (Press CONSOLE button to exit)
17 GR. MANY (Try it in full screen)
2 GR. 2 PINK 8 SE. MANY (Use SE. to change color 2)
4 GOLD 8 SE. (Use SE. to change background color).
0 GR. (Go back to normal text screen.)
```

You can also see a quick demonstration by loading the Graphics Demo Program listed on screen 170. If it's not listed on screen 170, do an INDEX in the area of the Graphics routine screens you loaded recently. When you find the Graphics examples screen, load it. Then do FBOX. Take a look at the code and then at the Glossary to get the idea.

As in Atari Basic, adding 16 to the graphics mode you want to enter gives non-split screen, and adding 32 suppresses erase-on-setup of the mode.



SOUNDS

As a final stop on this tour, load the SOUNDS words. The word SOUND acts similarly to the Basic command SOUND. In valFORTH it also has the abbreviation SO. and expects stack arguments like so:

channel(0-3) frequency#(0-255) distortion(0-14 evens) volume (0-15).
(We use "CatFish Don't Vote" as a mnemonic).

Try, for instance 0 200 12 8 SO. and then turn it off with 0 XSND which just shuts off the indicated voice, 0. or XSND4 which quiets everything. More about sound generation by the Atari may be found in the "sound" section.

Logical Line Input

One of the nice features of the Atari OS is that it lets you back the cursor over code that you've typed in already, even edit it with various inserts, deletes, and retypes, and then hit return to have it reinterpreted. This function is supported by valFORTH, and you can re-input up to two full lines of text, (and a wee bit more) at a time just by moving the cursor onto the "logical line" you wish to re-read. Try it.

THE GREAT SCREEN SIZE DEBATE

The "standard" Forth screen is composed of 1024 bytes. This is a nice round number, and on a good text display one can have room for that many characters plus a few more. However, beyond tradition, there is very little functional reason to have 1024 byte screens over several other power-of-2 sizes. In the case of Atari and Apple machines, 512 byte screens make video display editors much easier to work with, since one can get a whole screen in the display at once. valFORTH supports both 1024 and 512 byte screen modes, but in-house at Valpar we strongly prefer 512 byte screens and recommend that you adopt this as your personal standard. If at any time you wish to change to 1K to help compile software written on 1K screens, you can do so with one word, FULLK.

SAVING YOUR FAVORITE SYSTEM(S)

Well, you've seen many of the bells and whistles of valFORTH. When you are using the language for software development you will probably have a favorite set of capabilities that you always want aboard. Rather than loading them from scratch each time, why not SAVE them to a formatted disk? Just get everything you want into the dictionary. After it's all loaded, put a formatted disk into drive 1 and type SAVE. Answer the prompt by pressing "Y" unless you have changed your mind, and the computer will save a bootable copy of your system dictionary on the blank disk.

DISTRIBUTING YOUR PROGRAMS

If you have a program you wish to distribute, there are two ways in which to proceed:

(1) Make a PROTECTED auto-booting copy of your software by using the word AUTO as detailed in the "compiling Auto-Booting Software" section of this manual.

(2) Make a TARGET-COMPILED version of your software, using the valFORTH Target Compiler, scheduled for release approximately 9/82. Target Compilers allow production of much smaller final FORTH products by allowing elimination of unnecessary code, e.g., headers, compiler, buffers, etc.

In addition to the above procedures, Valpar International also requires that the message:

> Created in whole or part using valFORTH products of
> Valpar International, Tucson, AZ 85713, USA
> Based on fig-FORTH, provided through the courtesy of
> Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070

Hope you've enjoyed the tour. Bye now.

FIG EDITOR USER MANUAL

Based on the Manual
by Bill Stoddart
of FIG, United Kingdom

valFORTH organizes its mass storage into "screens" of 512 characters, with the option of 1024. If, for example, a diskette of 90K byte capacity is used entirely for storing text, it will appear to the user as 180 screens numbered 0 to 179. Screen 0 should not be used for program code. Each screen is organized as 16 lines with 32 characters per line.

Selecting a Screen and Input of Text

To start an editing session the user types EDITOR to invoke the appropriate vocabulary.

The screen to be edited is then selected, using either:

```
n LIST  ( list screen n and select it for editing ) OR
n CLEAR ( clear screen n and select for editing )
```

To input new text to screen n after LIST or CLEAR the P (put) command is used.

Example:

```
0 P THIS IS HOW
1 P TO INPUT TEXT
2 P TO LINES 0, 1, AND 2 OF THE SELECTED SCREEN.
```

Based on material provided through the courtesy of the FORTH INTEREST GROUP, P.O. Box 1105, San Carlos, CA 94070.

Line Editing

During this description of the editor, reference is made to PAD. This is a text buffer which may hold a line of text used by or saved with a line editing command, or a text string to be found or deleted by a string editing command.

PAD can be used to transfer a line from one screen to another, as well as to perform edit operations within a single screen.

Line Editor Commands

n H	Hold line n at PAD (used by system more often than by user).
N D	Delete line n but hold it in PAD. Line 15 becomes blank as lines n+1 to 15 move up 1 line.
n T	Type line n and save it in PAD.
n R	Replace line n with the text in PAD.
n I	Insert the text from PAD at line n, moving the old line n and following lines down. Line 15 is lost.
n E	Erase line n with blanks.
n S	Spread at line n. n and subsequent lines move down 1 line. Line n becomes blank. Line 15 is lost.

Cursor Control and String Editing

The screen of text being edited resides in a buffer area of storage. The editing cursor is a variable holding an offset into this buffer area. Commands are provided for the user to position the cursor, either directly or by searching for a string of buffer text, and to insert or delete text at the cursor position.

Commands to Position the Cursor

TOP	Position the cursor at the start of the screen.
N M	Move the cursor by a signed amount n and print the cursor line. The position of the cursor on its line is shown by a ● (solid circle).

String Editing Commands

F text	Search forward from the current cursor position until string "text" is found. The cursor is left at the end of the text string, and the cursor line is printed. If the string is not found an error message is given and the cursor is repositioned at the top of screen.
B	Used after F to back up the cursor by the length of the most recent text.
N	Find the next occurrence of the string found by an F command.
X text	Find and delete the string "text."
C text	Copy in text to the cursor line at the cursor position.
TILL text	Delete on the cursor line from the cursor till the end of the text string "text."
NOTE:	Typing C with no text will copy a null (represented by a heart) into the text at the cursor position. This will abruptly stop later compiling! To delete this error type TOP X 'return'.

Screen Editing Commands

n LIST	List screen n and select it for editing
n CLEAR	Clear screen n with blanks and select it for editing
n1 n2 COPY	Copy screen n1 to screen n2.
L	List the current screen. The cursor line is relisted after the screen listing, to show the cursor position.
FLUSH	Used at the end of an editing session to ensure that all entries and updates of text have been transferred to disc.

TEXT c ---
Accept following text to pad. c is text delimiter.

LINE n --- addr
Leave address of line n of current screen. This address will be in the disc buffer area.

WHERE n1 n2 ---
n2 is the block no., n1 is offset into block. If an error is found in the source when loading from disc, the recovery routine ERROR leaves these values on the stack to help the user locate the error. WHERE uses these to print the screen and line nos. and a picture of where the error occurred.

R# --- addr
A user variable which contains the offset of the editing cursor from the start of the screen.

#LOCATE --- n1 n2
From the cursor position determine the line-no n2 and the offset into the line n1.

#LEAD --- line-address offset-to-cursor

#LAG --- cursor-address count-after-cursor-till-EOL

-MOVE addr line-no ---
Move a line of text from addr to line of current screen.

H n ---
Hold numbered line at PAD.

E n ---
Erase line n with blanks.

S n ---
Spread. Lines n and following move down. n becomes blank.

D n ---
Delete line n, but hold in pad.

M n ---
Move cursor by a signed amount and print its line.

T n ---
Type line n and save in PAD.

L ---
List the current screen.

```
R       n ---
```
Replace line n with the text in PAD.

```
        n ---
```
Put the following text on line n.

```
I       n ---
```
Spread at line n and insert text from PAD.

```
TOP     ---
```
Position editing cursor at top of screen.

```
CLEAR   n ---
```
Clear screen n, can be used to select screen n for editing.

```
FLUSH   ---
```
Write all updated buffers to disc.

```
COPY    n1 n2 ---
```
Copy screen n1 to screen n2.

```
-TEXT   Addr 1 count Addr 2 -- boolean
```
True if strings exactly match.

```
MATCH   cursor-addr bytes-left-till-EOL str-addr str-count
 ---    tf cursor-advance-till-end-of-matching-text
 ---    ff bytes-left-till-EOL
```
Match the string at str-addr with all strings on the cursor line forward from the cursor. The arguments left allow the cursor R# to be updated either to the end of the matching text or to the start of the next line.

```
1LINE   --- f
```
Scan the cursor line for a match to PAD text. Return flag and update the cursor R# to the end of matching text, or to the start of the next line if no match is found.

```
FIND    ---
```
Search for a match to the string at PAD, from the cursor position till the end of screen. If no match found issue an error message and reposition the cursor at the top of screen.

```
DELETE  n ---
```

Delete n characters prior to the cursor.

```
N       ---
```
Find next occurrence of PAD text.

```
F       ---
```
Input following text to PAD and search for match from cursor position till end of screen.

B ---

Backup cursor by text in PAD.

X ---

Delete next occurrence of following text.

TILL ---

Delete on cursor line from cursor to end of the following text.

C ---

Spread at cursor and copy the following text into the cursor line.

RELOCATING BUFFERS

The purpose of this section is to show you how to avoid incorporating buffer space into an auto-booting program, thereby saving more than 2K in memory requirement for the machine on which the program will eventually run.

Fig-FORTH (and so valFORTH) uses a virtual memory arrangement which allows disk areas to be accessed in a manner similar to that used to access semiconductor memory. We won't go into detail here; those wishing to find out more about this can contact FIG for documentation at:

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

or they can puzzle out the process by starting at the word BLOCK. For our present purposes, however, we simply note that the virtual memory scheme requires that some continuous area of memory be allotted as buffer space for disk operation. valFORTH as delivered has buffer space for four 512 byte "screens" at a time. Each screen is composed of four blocks of 132 bytes each: 128 bytes of actual data, corresponding to a sector, and four bytes of identification and delimiting data. This produces a total of 4 x 4 x 132 = 2112 bytes that are needed for programming and compilation* but are generally not required when software is actually run. In order to get the full use of your computer, particularly for the purposes of producing auto-booting software like games, you'll need to know how memory is mapped and what changes you can make in the mapping. During the following discussion refer to the memory map provided with your documentation.

You will note from the memory map that the buffers are placed just above the kernel (boot-up) valFORTH dictionary. The dictionary pointer is set just past the buffers, so new word definitions will be compiled in above the end of the buffers. Why such an odd location? Read on...

* Those used to seeing the buffers at the top of memory will quickly realize that this is impractical on the Atari, since that area is used for display lists. Although it is possible to an extent to fool the operating system into thinking that it has less memory than it actually has, and thus "reserve" an area at the top of memory, this is a troublesome proposition.

* Another approach is to put the buffers just below the kernel dictionary, which has been done in at least one FORTH-for-Atari release. While this is safe, it sacrifices 2K bytes during run time unless rather clever programming techniques are used on each program to put code into the dormant buffer area.

* Clearly, the buffers should be put somewhere above the dictionary but below the display-list area, and a simple means to relocate them should be supported. This is precisely what you have in valFORTH.

* In a pinch, you can compile using only 264 bytes of buffer memory.

When you have a program that will compile and run, preferably without errors, and you'd like to create a smaller auto-booting version, follow this procedure:

* Boot the valFORTH disk.

* Decide on the area to which to relocate the buffers: If the program can be loaded without leaving the 0 Graphics mode or doing anything else to high memory while loading, then the result printed by the sequence

```
0 GR. DCX 741 @ 2113 - U  (See note below.)
```

will be a safe place to put the buffers; 741 @ is the Atari OS pointer to just below the current display list. (If you will be using Transients, a capability of the Utilities/Editor package, their default location is

```
DCX 741 @ 4000 -
```

so you would be better off to put the buffers at, say,

```
DCX 741 @ 6113 -
```

to avoid conflict).

* Find the buffer relocation utility listed in the table of contents starting on screen 170 of the valFORTH disk, and load it. This is a self-prompting utility that directs you to relocate the buffers and then forget the utility. Follow the directions. You'll receive a verification message after the buffers have been moved.

* Type

```
' TASK DP !
```

to move the dictionary pointer below the old buffer area. (Advanced programmers: This is not a typo. The cfa of TASK points to NEXT.)

* Now load your program as usual. You should probably create an auto-booting program at this point, rather than doing anything else, since if you run the program now it may write into your relocated buffers and conceivably even attempt a write to your disk. So, create an auto-booting version as directed in the Auto-booting section above. Remember that if the program is for distribution, you MUST protect your software and ours by using the AUTO command.

*****CAUTION*****

The buffers start out just above the kernel dictionary, as indicated, and for normal programming they should be LEFT THERE: Several routines on the valFORTH disk and other disks in this product line use the area between pad and the bottom of the display list as a scratch area for extensive disk transfers. DISKCOPY1 and DISKCOPY2 on the valFORTH disk are examples.

Note: The buffers should generally be relocated to an even address because of an Atari OS bug. See also Note 1 at end of valFORTH 1.1 Glossary.

COMPILING AUTO-BOOTING SOFTWARE

Your purchase of valFORTH and its associated packages also grants you a single-user license for the software. You may not copy valFORTH or its associated Valpar International products for any purpose other than for your own use as back-up copies. However, a word called AUTO has been provided to allow you to create a copy of your software that is suitable for distribution. The word AUTO does several things.

* AUTO provides extensive protection both for your software and the valFORTH and auxiliary programs on which it is based. Your product may still be copied by normal methods, but the programming concepts on which it is based will be very difficult to analyze. The valFORTH and auxiliary programs will be rendered useless except to run your program. Since AUTO scrambles all headers in the code before saving to disk or cassette, even direct examination of the code on the medium is not very revealing. This provides essentially all the protection of headerless code.

* AUTO will create a disk that autoboots to the FORTH word of your choice. This usually will be the last word defined in your program. In addition, a disk created using AUTO will not have exit points: That is, even if your program terminates, or makes an error because of an undiscovered bug, it will not exit to valFORTH and the "ok" prompt. Instead, it will automatically attempt to start again at the original auto-boot word, and will do so unless an error has disabled the system.

* AUTO allows repetitive saving of your protected software to disk and cassette in one sitting, with extensive prompting. This provides a short-run production environment. (Remember that if you want to save to cassette, the cassette recorder should be attached to the system at boot time; if it is attached after booting, the computer may not know that the recorder is there and may fail when trying to AUTO to cassette).

To run AUTO and create your bootable software:

(1) Load valFORTH.
(2) Relocate buffers to save 2K+, if desired (see below).
(3) Load your program.
(4) DISPOSE transients, if you use them. (The Transient utilities come with the Utilities/Editor package, and allow use of "disposable assemblers" and the like).
(5) Find the Auto-Boot Utility section on the valFORTH disk by referring to the directory starting on screen 170, and load as indicated.
(6) Type AUTO cccc where "cccc" is the word which you wish to execute on auto-booting the software. You will now be prompted through the rest of the procedure. On exiting from AUTO you will fall through to the auto-booting program that you have just protected.

DISTRIBUTING YOUR PROGRAMS

If you have a program you wish to distribute, there are two ways in which to proceed:

(1) Make a PROTECTED auto-booting copy of your software by using the word AUTO as detailed in the "Compiling Auto-Booting Software" section of this manual.

(2) Make a TARGET-COMPILED version of your software, using the valFORTH Target Compiler, scheduled for release approximately 9/82. Target Compilers allow production of much smaller final FORTH products by allowing elimination of unnecessary code, e.g., headers, compiler, buffers, etc.

In addition to the above procedures, Valpar International also requires that the message:

> Created in whole or part using valFORTH products of
> Valpar International, Tucson, AZ 85713, USA
> Based on fig-FORTH, provided through the courtesy of
> Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070

be included either on the outside of the media (diskette, cassette, or other) as distributed, or in the documentation provided with the product. Please note that failure to include this message with products that include valFORTH code may be regarded as a copyright violation.

GRAPHICS, COLORS, AND SOUNDS

Graphics

The Graphics package follows the Atari BASIC graphics set as closely as possible, and is identical in most respects. As in BASIC, the most complex parts of Graphics are DRAWTO (abbreviated "DR.") and FIL, and even these are not too obscure. Find the Graphics Demo by looking at the directory starting on screen 170, and load it. Try the word FBOX. Now look at the code that produced this effect, if you like. The general explanation is as follows:

Display positions are denoted by two coordinates, a horizontal and a vertical. The 0,0 point is in the upper left hand corner, and the vertical coordinate increases as you go down the display, while the horizontal coordinate increases as you go to the right. This is all familiar from BASIC.

In graphics modes, a single point at position X Y can be plotted by X Y PLOT. The color of the point will be that in the color register declared by the last COLOR command. A line, again of the color in the register declared by the last color command, may then be drawn to point X1 Y1 by X1 Y1 DR. . The word FIL may be used to fill in an area as described in the Atari manual, and as illustrated in the FBOX example. The color register for the fill is the one whose number is on the stack when FIL is executed. Essentially, to set up FIL you draw in boundaries and pick two points you wish to FIL between. The first of these points is set up either by a DR. or PLOT command, or by valFORTH's POSIT command. POSIT has the advantage of not requiring that you put anything into the place where you are positioning yourself. The second point for the FIL command is then set up by using POS. . The fill is then performed by putting a number on stack (the color register for the fill) and then doing FIL.

If you are in a text mode, a single character, c , can be sent to the display by ASCII c CPUT. Text strings can be sent to the display with G" cccc " and in addition will have the color in the register specified by the last COLOR command before the string is output. This is a significant enhancement to BASIC.

Graphics and Color Glossary:

SETCOLOR n1 n2 n3 --
Color register n1 (0...3 and 4 for background) is set to hue n2 (0 to 15) and luminance n3 (0-14, evens).

SE. n1 n2 n3 --
Alias for SETCOLOR.

GR. n --
Identical to GR. in BASIC. Adding 16 will suppress split display. Adding 32 will suppress display preclear. In addition, this GR. will not disturb player/missiles.

POS. x y --
Same as BASIC POSITION or POS. Positions the invisible cursor if in a split display mode, and the text cursor if in 0 GR. .

POSIT x y --
Positions and updates the cursor, similar to PLOT, but without changing display data.

PLOT x y --
Same as BASIC PLOT. PLOTs point of color in register specified by last COLOR command, at point x y.

DRAWTO x y --
Same as BASIC DRAWTO. Draws line from last PLOT'ted, DRAWTO'ed or POSIT'ed point to x y, using color in register specified by last COLOR command.

DR. x y --
Alias for DRAWTO.

FIL b --
Fills area between last PLOT'ed, DRAWTO'ed or POSIT'ed point to last position set by POS., using the color in register b.

G" --
Used in the form G" ccccc". Sends text cccc to text area in non-0 Graphics mode, starting at current cursor position, in color of register specified by last COLOR command prior to cccc being output. G" may be used within a colon definition, similar to .".

GTYPE addr count --
Starting at addr, output count characters to text area in non-0 Graphics mode, starting at current cursor position, in color of register specified by last COLOR command.

LOC. x y -- b
Positions the cursor at x y and fetches the data from display at that position. Like BASIC LOCATE and LOC. . Note that since the word LOCATE has a different meaning in valFORTH (it is part of the advanced editor in the Utilities/Editor package), the name is not used in this package. (Advanced users: We could put Graphics in its own vocabulary, but this would add some inconvenience.)

(G") --
Run-time code compiled in by G".

POS@ -- x y
Leaves the x and y coordinates of the cursor on the stack.

CPUT b --
Outputs the data b to the current cursor position.

CGET -- b
Fetches the data b from the current cursor position.

>SCD c1 -- c2
Converts c1 from ATASCII to its display screen code, c2.
Example: ASCII A >SCD 88 @ C!
will put an "A" into the upper left corner of the display.

SCD> c1 -- c2
Converts c1 from display screen code to ATASCII c2.
See >SCD.

>BSCD addr1 addr2 count --
Moves count bytes from addr1 to addr2, translating from ATASCII to display screen code on the way.

BSCD> addr1 addr2 count --
Moves count bytes from addr1 to addr2, translating from display screen code to ATASCII on the way.

COLOR b --
Saves the value b in the variable COLDAT.

CLRBYT -- addr
Variable that holds data from last COLOR command.

GREY -- 0
GOLD -- 1
ORNG -- 2
RDORNG -- 3
PINK -- 4
LVNDR -- 5
BLPRPL -- 6 (CONSTANTs)
PRPLBL -- 7
BLUE -- 8
LTBLUE -- 9
TURQ -- 10
GRNBL -- 11
GREEN -- 12
YLWGRN -- 13
ORNGRN -- 14
LTORNG -- 15

BOOTCOLOR hue lum --
Sets up hue for playfield 2 (text background) and lum for playfield 1 (letter intensity) in 0 Graphics mode. Lum of playfield 2 is set at 4.
After using BOOTCOLOR, doing SAVE will create a system disk with the selected color.

Sounds

The actual production of sound by the Atari machines is rather complex and the reader is referred to the many recent (first half 1982) articles on this subject in various magazines. Here we will restrict comments to the function of the Atari audio control register. This is an eight bit register which valFORTH shadows by the variable AUDCTL. The bits have the following functions:

bit 7: Change 17 bit polycounter to 9 bit polycounter.
Affects distortions 0 and 8.
bit 6: Clock channel 0 with 1.79 Mhz instead of 64 Khz.
bit 5: Clock channel 2 with 1.79 Mhz instead of 64 Khz.
bit 4: Clock channel 1 with channel 0 instead of 64 Khz.
bit 3: Clock channel 3 with channel 2 instead of 64 Khz.
bit 2: Use channel 2 as crude high-pass on channel 0.
bit 1: Use channel 3 as crude high-pass on channel 1.
bit 0: Change normal 64 Khz to 15 Khz.

The value n may be sent to the audio control register by doing n FILTER!.

SOUND chan freq dist vol --
Sets up the sound channel "chan" as indicated.
Channel: 0-3.
Frequency: 0-255, 0 is highest pitch.
Distortion: 0-14, evens only.
Volume: 0-15.
Suggested mnemonic: CatFish Don't Vote

SO. chan freq dist vol --
Alias of SOUND.

FILTER! n --
Stores n in the audio control register and into the valFORTH shadow register, AUDCTL. Use AUDCTL when doing bit manipulation, then do FILTER!. (FILTER! does a number of housekeeping chores, so use it instead of a direct store into the hardware register.)

AUDCTL -- addr
A variable containing the last value sent to the audio control register by FILTER!. Used for bit manipulation since the audio control register is write-only.

XSND n --
Silences channel n.

XSND4 --
Silences all channels.

S: flag --
If flag is true, enables handler that sends text to text screen. If false, disables the handler. (See PFLAG in main glossary.) ON S: etc.

P : flag --
If flag is true, enables handler that sends text to printer. If false, disables the handler. (See PFLAG in main glossary.) OFF P: etc.

BEEP --
Makes a raucous noise from the keyboard. Is put in this package for lack of a better place.

ASCII c, -- n (executing)
c, -- (compiling)
Converts next character in input stream to ATASCII code. If executing, leaves on stack. If compiling, compiles as literal.

EJECT --
Causes a form feed on smart printers if the printer handler has been enabled by ON P:. May need adjustment for dumb or nonstandard printers.

LISTS start count --
From start, lists count screens. May be aborted by CONSOLE button at the end of a screen.

PLIST scr --
Lists screen scr to the printer, then restores former printer handler status.

PLISTS start cnt --
From start, lists cnt screens to printer three to a page, then restores former printer handler status. May be aborted by CONSOLE button at the end of a screen.

FORMAT --
With prompts, will format a disk in drive of your choice.

(FMT) n1 -- n2
Formats disk in drive n1. Leaves 1 for good format, otherwise error number. Note: Because of what appears to be an OS peculiarity, this operation must not be the first disk access after a boot.

DISKCOPY1 --
With prompts, copies a source to a destination disk on single drive, with swapping. Smart routine uses all memory from PAD to bottom of Display List, producing minimum number of swaps.

DISKCOPY2 --
With prompts, copies disk in drive 1 to disk in drive 2 using memory like DISKCOPY1.

DECOMP cccc
Does a decompilation of the word cccc if it can be found in the active vocabularies.
Although DECOMP is very smart, like most FORTH decompilers it will become confused by certain constructs, and will begin to print trash, with pauses in between while it looks for more trash to print. When this happens, simply hold down a CONSOLE button until DECOMP exits. This sometimes takes as much as 10 seconds, depending on luck.

CDUMP addr n --
A character dump from addr for at least n characters. (Will always do a multiple of 16.)

#DUMP addr n --
A numerical dump in the current base for at least n characters. (Will always do a multiple of 8.)

(FREE) -- n
Leaves number of bytes between bottom of display list and PAD. This is essentially the amount of free dictionary space, if additional memory is not being used for player/missiles, extra character sets, and so on.

FREE --
Does (FREE) and then prints the stack and "bytes".

H. n --
Prints n in HEX, leaves BASE unchanged.

STACK flag --
If flag is true, turns on visible stack. If flag is false, turns off visible stack.

.S ... -- ...
Does a signed, nondestructive stack printout, TOS at right. Also sets visible stack to do signed printout.

U.S ... -- ...
Does unsigned, nondestructive stack printout, TOS at right. Also sets visible stack to do unsigned printout.

B? --
Prints the current base, in decimal. Leaves BASE undisturbed.

CFALIT cccc, -- cfa (executing)
cccc, -- (compiling)
Gets the cfa (code field address) of cccc. If executing, leaves it on the stack; if compiling, compiles it as a literal. Not precisely a debugging tool, but finds use in DECOMP.

The floating-point package uses the Atari floating point routines in the operating system ROM in the same way that Atari Basic does. The routines are rather slow, and there are no trigonometric functions internal to the Atari. (SIN, COS, TAN, ATN, and ATN2 have been programmed and are available in the Advanced Graphics/Floating Point Package.) LOG and EXP are included in the operating system ROM and are supported in the present package, in base 10 and base e. Note that in the directory on screen 170 it is indicated that the ASSEMBLER must be loaded before loading the floating-point package.

Floating point words have a six byte representation in the Atari OS, and since the stack has a 60 byte maximum, a maximum of 10 floating point numbers can be on the stack at a time. In practice, this maximum often becomes 9 since some fp routines use the stack as a scratch area.

Operations involving floating-point numbers generally leave floating-point results. Exceptions are the words FIX, which takes a positive floating pointer number less than 32767.5 and leaves a rounded integer; and the floating-point comparison operators, F=, F<, etc., which leave flags. To get a floating-point number on the stack, use the word FLOATING or its alias, FP, followed by a number in Fortran "E" format. For example,

```
        FP  12345
        FP  12345.6
        FP  -12345.8
        FP  +5432E-16
  and   FP  -8E18
```

will all leave floating-point numbers on the stack. Floating-point variables and constants are also supported.

It has been our experience that mistakes are common when first using this package. One must remember to use F* and not *, F+ and not +, and so on, when doing fp operations. Remember also that integers and fp numbers can't be mixed by operations: Either convert the fp number by FIX, or the integer by FLOAT, and then use the appropriate operation.

Create new words as usual. For instance, to define a floating-point square root function, write

```
: FSQRT                  ( fp -- fp )
  LOG  FP  2  F/  EXP ;
```

Overflow and underflow, and illegal operations such as dividing by 0, taking logarithms of negative numbers, or FIXing a negative number cause undefined and rather unpredictable results, though they do not harm the system. (Additional words in the Utilities/Editor Package cause all but one of these operations to give correct or useable results; logarithms of negatives cannot be approximated with Real numbers.)

The maximum and minimum numbers are generous, about 1E97 and 1E-97, and it is sometimes possible to exceed these limits during computation. Atari's internal representation of floating point numbers is awkward. Refer to the Atari OS manual, available from Atari, for details if needed.

FLOATING-POINT GLOSSARY

In the following, "fp" is used to indicate a floating-point number (six bytes) on the stack. The terms "top-of-stack," "2nd-on-stack" etc., have been used with the obvious meanings even though, because fp numbers are six bytes, their physical positions on the stack will not match the usual ones.

FCONSTANT cccc, fp --
cccc: --fp
The character string is assigned the constant value fp. When cccc is executed, fp will be put on the stack.
Example: FP 3.1415926 FCONSTANT PI

FVARIABLE cccc, fp --
cccc: addr --
The character string cccc is assigned the initial value fp. When cccc is executed, the addr (two bytes) of the value of cccc will be put on the stack.
Example: FP 0 FVARIABLE X
FP 18.4 X F!

FDUP fp1 -- fp1 fp1
Copies the fp number at top-of-stack.

FDROP fp --
Discards the fp number at top-of-stack.

FOVER fp2 fp1 -- fp2 fp1 fp2
Copies the fp number at 2nd-on-stack to top-of-stack.

FLOATING cccc, -- fp
Attempts to convert the following string, cccc, to a fp number. Stops on reaching first unconvertible character and skips the rest of the string. If no characters convertible, leaves unpredictable fp number on stack.

FP cccc, --fp
Alias for FLOATING.

F@ addr -- fp
Fetches the fp number whose address is at top-of-stack.

F! fp addr --
Stores fp into addr. Remember that the operation will take six bytes in memory.

F. fp --
Type out the fp number at top-of-stack. Ignores the current value in BASE and uses base 10.

F? addr --
Fetches a fp number from addr and types it out.

F+ fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their fp sum, fp3.

F- fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their difference, fp3=fp2-fp1.

F* fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their product, fp3.

F/ fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their quotient, fp3=fp2/fp1.

FLOAT n -- fp
Replaces number at top-of-stack with its fp equivalent.

FIX fp (non-neg, less than 32767.5) -- n
Replaces fp number at top-of-stack, constrained as indicated, with its integer equivalent.

LOG fp1 -- fp2
Replaces fp1 with its base e logarithm, fp2. Not defined for fp1 negative.

LOG10 fp1 -- fp2
Replaces fp1 with its base 10 decimal logarithm, fp2. Not defined for fp1 negative.

EXP fp1 -- fp2
Replaces fp1 with fp2, which equals e to the power fp1.

EXP10 fp1--fp2
Replaces fp1 with fp2, which equals 10 to the power fp1.

F0= fp -- flag
If fp is equal to floating-point 0, a true flag is left. Otherwise, a false flag is left.

F= fp2 fp1 -- flag
If fp2 is equal to fp1, a true flag is left. Otherwise, a false flag is left.

F> fp2 fp1 -- flag
If fp2 is greater than fp1, a true flag is left. Otherwise, a false flag is left.

F< fp2 fp1 -- flag
If fp2 is less than fp1, a true flag is left. Otherwise, a false flag is left.

FLITERAL fp --
If compiling, then compile the fp stack value as a fp literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

```
: xxx [ calculate ] FLITERAL ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is resumed and FLITERAL compiles the value on stack.

FLIT -- fp
Within a colon definition, FLIT is automatically compiled before each fp number encountered as input text. Later execution by the system of FLIT as it is encountered in the dictionary cause the context of the next 6 dictionary addresses to be pushed to the stack as a fp number. FLIT is also compiled in explicitly by FLITERAL.

ASCF addr -- fp
An ASCII-to-floating-point conversion routine. Uses Atari OS routine. The routine reads string starting at addr and attempts to create a floating point number. If string is not a valid ASCII floating-point representation, leaves undefined result on stack. Used by FLOATING.

FS fp --
System routine. Sends fp argument on stack to Atari register FR0. Experts only.

>F -- fp
System routine. Fetches fp argument from Atari register FR0. Experts only.

<F fp1 fp2 --
System routine. Sends fp1 and fp2 to Atari registers FR1 and FR0 respectively. Experts only.

F.TY --
System routine. Types out last fp number converted by FASC.

CIX addr --
System variable. One byte offset pointer in buffer pointed to by INBUF. Experts only.

INBUF addr --
System variable. Used by ASCF to know where ASCII string to be converted is located.

FR1 -- n
System constant. Atari internal register address.

FR0 --n
System constant. Atari internal register address.

FPOLY addr count --
A system routine for advanced users doing polynomial evaluation. The polynomial P(Z) = SUM(i=0 to n) (A(i)*Z**i) is computed by the following standard method:
P(Z) = (...(A(n)*Z + A(n-1))*Z + ... + A(1))*Z + A(0)
The address addr points to the coefficients A(i) stored sequentially in memory, with the highest order coefficient first. The count is the number of coefficients in the list. The independent variable Z, in floating-point, should be sent to FR0 using FS. FPOLY is then executed. The result put on the stack using >F. Note that FPOLY is intended to be used in a Forth word. Trigonometric functions and general polynomial expansions, for example, may be defined more simply with the help of this routine.

FLG10 --
System routine used by LOG10.

FLG --
System routine used by LOG.

FEX --
System routine used by EXP.

FEX10 --
System routine used by EXP10.

FDIV --
System routine used by F/.

FMUL --
System routine used by F*.

FSUB --
System routine used by F-.

FADD --
System routine used by F+.

FPI --
System routine used by FIX.

IFP --
System routine used by FLOAT.

FASC --
System routine, Does floating-point-to-ASCII conversion on the fp number in FR0 and leaves string at address pointed to by INBUF. Last byte of string has most significant bit set. Used by F.TY.

AFP --
System routine used by ASCF.

(intentionally left blank)

OPERATING SYSTEM

This package implements the computer's Operating System I/O routines. The 850 (RS-232C) driver package may be loaded into the dictionary by using the word RS232, which will then support references to devices "R1" through "R4."

The code for this section was originally written by Patrick Mullarky, and published through the Atari Program Exchange. It is used here by permission of the author.

OS GLOSSARY

OPEN addr n1 n2 n3 -- n4
This word opens the device whose name is at addr. The device is opened on channel n3 with AUX1 and AUX2 as n1 and n2 respectively. The device status byte is returned as n4. The name of a device may be produced in various ways: For a single character name, say "S" for the screen handler,

```
ASCII S PAD C!
```

will leave the ASCII value of S at PAD. Then

```
PAD 8 0 3 OPEN
```

will open the screen handler on channel 3 with AUX1 = 8 (write only) and AUX2 = 0. If you have the UTILITIES/EDITOR Package, longer names may be set up simply by using the word " .

CLOSE n --
Closes channel n.

PUT b1 n -- b2
Outputs byte b1 on channel n, returns status byte b2.

GET n -- b1 b2
Gets byte b1 from channel n, returns status byte b2.

GETREC addr n1 n2 -- n3
Inputs record from channel n2 up to length n1. Returns status byte n3.

PUTREC addr n1 n2 -- n3
Outputs n1 characters starting at addr through channel n2. Returns status byte n3.

STATUS n -- b
Returns status byte b from channel n.

DEVSTAT n -- b1 b2 b3
From channel n1 gets device status bytes b1 and b2, and normal status byte b3.

SPECIAL b1 b2 b3 b4 b5 b6 b7 b8 -- b9
Implements the Operating System "Special" command. AUX1 through AUX6 are b1 through b6 respectively, command byte is b7, channel number is b8. Returns status byte b9.

RS232 --
Loads the Atari 850 drivers into the dictionary (approx 1.8K) through a three-step bootstrap process. Executing this command more than once without turning the 850 off and on again will crash the system.

valForth Glossary

Based on the fig-Forth Glossary
Provided through the courtesy of
Fourth Interest Group, P.O. Box 1105, San Carlos, CA 94070

This glossary contains all of the word definitions in Release 1.1 of valForth. The definitions are presented in the order of their ASCII sort.

The first line of each entry shows a symbolic description of the action of the procedure on the parameter stack. The symbols indicate the order in which input parameters have been placed on the stack. Two dashes "--" indicate the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address
b	8 bit byte (i.e. hi 8 bits zero)
c	7 bit ASCII character (hi 9 bits zero)
d	32 bit signed double integer, most significant portion with sign on top of stack.
f	boolean flag. 0=false, non-zero=true
tf	boolean true flag=non-zero
ff	boolean false flag=0
n	16 bit signed integer number
u	16 bit unsigned integer

The capital letters on the right show definition characteristics:

C	May only be used within a colon definition. A digit indicates number of memory addresses used, if other than one.
E	Intended for execution only.
L0	Level Zero definition of FORTH-78
L1	Level One definition of FORTH-78
P	Has precedence bit set. Will execute even when compiling. (immediate)
U	A user variable.
V	A valForth word not in fig-Forth.
B	A word adopted from Leo Brodie's Starting Forth.

Unless otherwise noted, all references to numbers are for 16 bit signed integers. The high byte of 16 bit numbers is the second byte on the stack, with the sign in the leftmost bit. For 32 bit signed double numbers, the most significant part (with the sign) is on top.

All arithmetic is implicitly 16 bit signed integer math, with error and underflow indication unspecified.

! n addr -- LO
Store 16 bits of n at address. Pronounced "store".

!CSP --
Save the stack position in CSP. Used as part of the compiler security.

d1 -- d2 LO
Generate from a double number d1, the next ASCII character which is placed in an output string. Result d2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #>. See #S. Pronounced "number".

#> d -- addr count LO
Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE. Pronounced "number-bracket".

#S d1 -- d2 LO
Generates ASCII text in the text output buffer, by the use of #, until a zero double number d2 results. Used between <# and #>. Pronounced "numbers".

' -- addr P,LO
Used in the form:
' nnnn
Leaves the parameter field address of dictionary word nnnn. As a compiler directive, executes in a colon-definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Pronounced "tick".

'(-- V,E,P
Used in the form:
'(WORD0 WORD1 . . . WORDN)(WORDN+1 . . .WORDM)
which executes as follows: If WORD0 is found in a search of CONTEXT and CURRENT, then execute WORD1 . . . WORDM. Generally used for conditional compilation. Note that if no words are to be included in the second group, then)(and) must be separated by at least TWO blanks.

(-- P,LO
Used in the form:
(cccc)
Ignore a comment that will be delimited by a right parenthesis on the same line. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required.

) -- V,E,P
No operation. Used in '(constructs.

)(-- V,E,P
Scans text input pointer past ")". Used in '(constructs.

(.") -- C+
The run-time procedure, compiled by ." which transmits the following in-line text to the selected output device. See ."

(/LOOP) n -- B,C2
Execution time code of /LOOP.

(;CODE) -- C
The run-time procedure, compiled by ;CODE, that rewrites the code field of the most recently defined word to point to the following machine code sequence. See ;CODE.

(+LOOP) n -- C2
The run-time procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(ABORT) --
Executes after an error when WARNING is -1. This word normally executes ABORT, but may be altered (with care) to a user's alternative procedure.

(DO) -- C
The run-time procedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) addr1 addr2 -- pfa b tf (ok)
addr1 addr2 -- ff (bad)
Searches the dictionary starting at the name field address addr2, matching to the text at addr1. Returns parameter field address, length byte of name field and boolean true for a good match. If no match is found, only a boolean false is left.

(FMT) n1 -- n2 V
Formats disk in drive n1. Leaves 1 for good format, otherwise error number. Note: Because of what appears to be an OS peculiarity, this operation must not be the first disk access after a boot.

(LINE) n1 n2 -- addr count
Convert the line number n1 and the screen n2 to the disc buffer address containing the data.

(LOOP) -- C2
The run-time procedure compiled by LOOP which increments the loop index and tests for loop completion. See LOOP.

(NUMBER) d1 addr1 -- d2 addr2
Convert the ASCII text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. Addr2 is the address of the first unconvertible digit. Used by NUMBER.

(SAVE) -- V
Used by SAVE, not generally used in programs. Sets up various parameters preparatory to writing a bootable disk.

* n1 n2 -- prod LO
Leave the signed product of two signed numbers. Pronounced "star".

*/ n1 n2 n3 -- n4 LO
Leave the ratio n4 = n1*n2/n3 where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence:
n1 n2 * n3 /
Pronounced "star-slash".

*/MOD n1 n2 n3 -- n4 n5 LO
Leave the quotient n5 and remainder n4 of the operation n1*n2/n3. A 31 bit intermediate product is used as for */. Pronounced "star slash-mod".

+ n1 n2 -- sum LO
Leave the sum n1+n2.

+! n addr -- LO
Add n to the value at the address. Pronounced "plus-store".

+- n1 n2 -- n3
Apply the sign of n2 to n1, which is left as n3.

+BUF addl -- addr2 f
Advance the disc buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV.

+LOOP n1 -- (run)
addr n2 -- (compile) P,C2,LO
Used in a colon-definition in the form:
DO . . . n1 +LOOP
At run-time, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The signed increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1>0), or until the new index is equal to or less that the limit (n1<0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

+ORIGIN n -- addr
Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This definition is used to access or modify the boot-up parameters at the origin area.

, n -- LO
Store n into the next available dictionary memory cell, advancing the dictionary pointer. (comma)

- n1 n2 -- diff LO
Leave the difference of n1-n2.

--> -- P,LO
Continue interpretation with the next disc screen. Pronounced "next-screen".

-DISK addr n2 n3 flag -- n4 V
Used by R/W. Not generally used in programs. This word performs a single-sector read or write on a disk. Addr is the starting RAM address, n2 is the sector number (1-720), n3 is the drive number (1-4), and the flag is 1 for read and 0 for write. On return, n4 will be zero if there were no problems, or it will be a DOS error number if a DOS error occurred.

-DUP

n1 -- n1 (if zero)
n1 -- n1 n1 (non-zero) LO

Reproduce n1 only if it is non-zero. This is usually used to copy a value just before IF, to eliminate the need for an ELSE part to drop it.

-FIND

-- pfa b tf (found)
-- ff (not found)

Accepts the next text word (delimited by blanks) in the input stream to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true are left. Otherwise, only a boolean false is left.

-TRAILING

addr n1 -- addr n2

Adjusts the character count n1 of a text string beginning address to suppress the output of trailing blanks. i.e. the characters at addr+n1 to addr+n2 are blanks.

.

n -- LO

Print a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing blanks follows. Pronounced "dot".

."

P,LO

Used in the form:

." cccc"

Compiles an in-line string cccc (delimited by the trailing ") with an execution procedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final ". The maximum number of characters may be an installation dependent value. See (.").

.LINE

line scr --

Print on the terminal device, a line of text from the disc by its line and screen number. Trailing blanks are suppressed.

.R

n1 n2 --

Print the number n1 right aligned in a field whose width is n2. No following blank is printed.

/

n1 n2 -- quot LO

Leave the signed quotient of n1/n2.

/LOOP

n -- B,C2

Like +LOOP, but uses an unsigned limit, index, and increment. Thus loop index may pass $7FFF without mishap. Faster than +LOOP and may be used instead of +LOOP if increment is positive (and index doesn't cross $7FFF.)

/MOD

n1 n2 -- rem quot LO

Leave the remainder and signed quotient of n1/n2. The remainder has the sign of the dividend.

0 1 2 3 -- n

These small numbers are used so often that it is attractive to define them by name in the dictionary as constants.

0# n -- flag V

Leaves a true flag if n is not equal to 0. Otherwise, leaves a false flag. Pronounced "zero not equal".

0< n -- f LO

Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0= n -- f LO

Leave a true flag if the number is equal to zero, otherwise leave a false flag.

0> n -- flag V

Leaves a true flag if n is greater than 0. Otherwise leaves a false flag.

OBRANCH f -- C2

The run-time procedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+ n1 -- n2 L1

Increment n1 by 1.

1- n1 -- n2 B

Subtract one from n1. Pronounced "one-minus".

2* n1 -- n2 B

Multiply n1 by two. Pronounced "two-star" or "two-times".

2+ n1 -- n2

Leave n1 incremented by 2.

2- n1 -- n2

Leave n1 decremented by 2.

2/ n1 -- n2 B

Divide n1 by two. Pronounced "two-slash".

2DROP d -- B

Drops the double number at TOS.

2DUP d -- d d B

Copies double number at TOS.

2OVER d2 d1 -- d2 d1 d2 B

Copies double number at 2OS to TOS.

2ROT d3 d2 d1 -- d2 d1 d3 V

Moves double number at 3OS over two double numbers on 2OS and TOS.

2SWAP d2 d1 -- d1 d2 B

Exchanges double numbers at TOS and 2OS.

: -- P,E,LO

Used in the form called a colon-definition:

: cccc ... ;

Creates a dictionary entry defining cccc as equivalent to the following sequence of Forth word definitions '...' until the next ';' or ';CODE'. The compiling process is done by the text interpreter as long as STATE is non-zero. Other details are that the CONTEXT vocabulary is set to the CURRENT vocabulary and that words with the precedence bit set (P) are executed rather than being compiled.

; -- P,C,LO

Terminate a colon-definition and stop further compilation. Compiles the run-time ;S.

;CODE -- P,C,LO

Used in the form:

: cccc .... ;CODE assembly mnemonics

Stop compilation and terminate a new defining word cccc by compiling (;CODE). Set the CONTEXT vocabulary to Assember, assembling to machine code the following mnemonics.

When cccc later executes in the form:

cccc nnnn

the word nnnn will be created with its execution procedure given by the machine code following cccc. That is, when nnnn is executed, it does so by jumping to the code after nnnn. An existing defining word must exist in cccc prior to ;CODE.

;S -- P,LO

Stop interpretation of a screen. ;S is also the run-time word compiled at the end of a colon-definition which returns execution to the calling procedure.

< n1 n2 -- f LO

Leave a true flag if n1 is less than n2; otherwise leave a false flag.

<# -- LO

Set-up for pictured numeric output formatting using the words:

<# # #S SIGN #>

The conversion is done on a double number producing text at PAD. Pronounced "Bracket Number".

<= n2 n1 -- flag V

Leaves true flag if n2 is less than or equal to n1. Otherwise, leaves false flag.

<> n2 n1 -- flag V

Leaves true flag if n2 and n1 are unequal. Otherwise, leaves false flag.

<BUILDS -- C,LO

Used within a colon-definition:

```
: cccc <BUILDS ...
       DOES>   ...  ;
```

Each time cccc is executed, <BUILDS defines a new word with a high-level execution procedure. Executing cccc in the form:

```
cccc nnnn
```

uses <BUILDS to create a dictionary entry for nnnn with a call to the DOES> part for nnnn. When nnnn is later executed, it has the address of its parameter area on the stack and executes the words after DOES> in cccc. <BUILDS and DOES> allow run-time procedures to be written in high-level rather than in assembler code (as required by ;CODE).

= n1 n2 -- f LO

Leave a true flag if n1=n2; otherwise leave a false flag.

==> --

If using 512 byte (half-K) screen, does --> otherwise, no action is taken. Used to chain screens in half-K format that will still load correctly in valFORTH full-K format.

> n1 n2 -- f LO

Leave a true flag if n1 is greater than n2; otherwise a false flag.

>= n2 n1 -- flag V

Leaves true flag if n2 is greater than or equal to n1. Otherwise, leaves false flag.

>R n -- C,LO

Remove a number from the computation stack and place as the most accessable on the return stack. Use should be balanced with R> in the same definition.

? addr -- LO

Print the value contained at the address in free format according to the current base.

?1K -- flag V

Leaves a true flag if C/L is 64, indicating 1K screens. Otherwise, leaves a false flag.

?COMP --

Issue error message if not compiling.

?CSP --

Issue error message if stack position differs from value saved in CSP.

?ERROR f n --

Issue an error message number n, if the boolean flag is true.

?EXEC --

Issue an error message if not executing.

?EXIT -- V,C

Caution: Use only within a DO LOOP. Within DO LOOP, will cause exit at end of current loop if a CONSOLE button is depressed when ?EXIT is

executed. Will work only in the word in which the DO LOOP is defined, not in a word nested further down.

?LOADING --

Issue an error message if not loading.

?PAIRS n1 n2 --

Issue an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK --

Issue an error message if the stack is out of bounds. This definition may be installation dependent.

?TERMINAL -- b

Perform a test of the terminal keyboard for actuation of a CONSOLE key. Leaves 0 if none actuated, leaves 1 for START, 2 for SELECT, 4 for OPTION, and sums for combinations.

@ addr -- n LO

Leave the 16 bit contents of address.

@EX addr -- V

Fetches the word (presumably a code field address) at addr, and then causes it to execute. Used for conditional execution without the speed and memory loss of flags and/or case statements. Typical use would be AUXOP @EX where the variable AUXOP had been loaded with the cfa of the desired word, by ' DESIREDWORD CFA AUXOP ! . Pronounced "fetch-ex."

ABORT -- LO

Clear the stacks and enter the execution state. Return control to the operators terminal, printing a message appropriate to the installation.

ABS n -- u LO

Leave the absolute value of n as u.

ACCEPT addr count --

Same as EXPECT, except that ACCEPT uses the O.S. line input routine which allows full MEMO PAD editing functions to be utilized. EXPECT prevents hazards such as Shift-Clear while ACCEPT does not. SEE EXPECT.

AGAIN addr n -- (compiling) P,C2,LO

Used in a colon-definition in the form:

BEGIN ... AGAIN

At run-time, AGAIN forces execution to return to corresponding BEGIN. There is no effect on the stack. Execution cannot leave this loop (unless R> DROP is executed one level below).

At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

ALLOT n -- LO

Add the signed number to the dictionary pointer DP. May be used to reserve dictionary space or re-origin memory.

AND n1 n2 -- n3 LO

Leave the bitwise logical and of n1 and n2 as n3.

B/BUF -- n

This constant leaves the number of bytes per disc buffer, the byte count read from disc by BLOCK.

B/SCR -- n

This constant leaves the number of blocks per editing screen.

BACK addr --

Calculate the backward branch offset from HERE to addr and compile into the next available dictionary memory address.

BASE -- addr U,LO

A user variable containing the current number base used for input and output conversion.

BEGIN -- addr n (compiling) P,LO

Occurs in a colon-definition in form:

```
BEGIN ... UNTIL
BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT
```

At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs.

At compile time, BEGIN leaves its return address and n for compiler error checking.

BL -- c

A constant that leaves the ASCII value for "blank".

BLANKS addr count --

Fill an area of memory beginning at addr with blanks.

BLK -- addr U,LO

A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

BLOCK n -- addr LO

Leave the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disc to which ever buffer was least recently written. If the block occupying that buffer has been marked as up-dated, it is rewritten to disc before block n is read into the buffer. See also BUFFER, R/W UPDATE FLUSH

BOOT -- V

Boots disk in drive 1. Same effect as turning computer on and off.

BRANCH -- C2,LO

The run-time procedure to unconditionally branch. An in-line offset is added to the interpretive pointer IP to branch ahead or back. BRANCH is compiled by ELSE, AGAIN, REPEAT.

BUFFER n -- addr

Obtain the next memory buffer, assigning it to block n. If the contents of the buffer is marked as updated, it is written to the disc. The block is not read from the disc. The address left is the first byte within the buffer for data storage.

C! b addr --

Store 8 bits at address.

C, b --

Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer.

C/L -- n V

A CONSTANT equal to the number of characters per line in the ValForth screen. Usually 32, but may be 64 if 1024 byte screens in use.

C? addr -- V

Fetches a byte from addr and prints it using . .

C@ addr --- b

Leave the 8 bit contents of memory address.

CFA pfa --- cfa

Convert the parameter field address of a definition to its code field address.

CMOVE from to count ---

Move the specified quantity of bytes beginning at address from to address to. The contents of address from is moved first proceeding toward high memory.

COLD --

The cold start procedure to adjust the dictionary pointer to the minimum standard and restart via ABORT. May be called from the terminal to remove application programs and restart.

COMPILE -- C2

When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).

CONSTANT n -- LO

A defining word used in the form:

n CONSTANT cccc

to create word cccc, with its parameter field containing n. When cccc is later executed, it will push the value of n to the stack.

CONTEXT -- addr U,LO

A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

COUNT addr1 -- addr2 n LO

Leave the byte address addr2 and byte count n of a message text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts with the second byte. Typically, COUNT is followed by TYPE.

CR -- LO

Transmit a carriage return and line feed to the selected output device.

CREATE --

A defining word used in the form:

```
CREATE  cccc
```

by such words as CODE and CONSTANT to create a dictionary header for a FORTH definition. The code field contains the address of the word's parameter field. The new word is created in the CURRENT vocabulary.

CSAVE -- V

Creates a bootable copy of RAM-resident system up to HERE on cassette. Computer beeps twice to indicate user must press Record and Play buttons on recorder, prior to pressing RETURN. CSAVE expects leaderless tape. If your tape has a leader, wind to just before the end of leader.

CSP -- addr U

A user variable temporarily storing the stack pointer position, for compilation error checking.

CURRENT -- addr

Address of a pointer to second word in the parameter field of the current vocabulary. (The current vocabulary is the one to which new definitions are added.)

D! d addr -- V

Stores double number d into addr.

D+ d1 d2 -- dsum

Leave the double number sum of two double numbers.

D+- d1 n -- d2

Apply the sign of n to the double number d1, leaving it as d2.

D. d -- L1

Print a signed double number from a 32 bit two's complement value. The high-order 16 bits are most accessable on the stack. Conversion is performed according to the current BASE. A blank follows. Pronounced "D-dot."

D.R d n --

Print a signed double number d right aligned in a field n characters wide.

D@ addr -- d V

Fetches double number d from addr.

DABS d -- ud

Leave the absolute value ud of a double number.

DECIMAL -- LO

Set the numeric conversion BASE for decimal input-output.

DEFINITIONS -- L1

Used in the form:

cccc DEFINITIONS

Set the CURRENT vocabulary to the CONTEXT vocabulary. In the example, executing vocabulary name cccc made it the CONTEXT vocabulary and executing DEFINITIONS made both specify vocabulary cccc.

DIGIT c n1 -- n2 tf (ok)
c n1 -- ff (bad)

Converts the ASCII character c (using base n1) to its binary equivalent n2, accompanied by a true flag. If the conversion is invalid, leaves only a false flag.

DLITERAL d -- d (executing)
d -- (compiling) P

If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.

DMINUS d1 -- d2

Convert d1 to its double number two's complement.

DO n1 n2 -- (execute)
addr n -- (compile) P,C2;LO

Occurs in a colon-definition in form:

DO ... LOOP
DO .. +LOOP

At run-time, DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP, the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after DO; otherwise, the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at run-time and may be the result of other operations. Within a loop "I" will copy the current value of the index to the stack. See I, LOOP, +LOOP, LEAVE.

When compiling within the colon-definition, DO compiles (DO), leaves the following address addr and n for later error checking.

DOES> -- LO

A word which defines the run-time action within a high-level defining word. DOES> alters the code field and first parameter of the new word to execute the sequence of compiled work addresses following DOES>. Used in combination with <BUILDS. When the DOES> part executes it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include the Forth assembler, multi-dimensional arrays, and compiler generation.

DP -- addr U,L

A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT, or directly.

DPL -- addr U,LO

A user variable containing the number of digits to the right of the decimal on double integer input. It may also be used hold output column location of a decimal point, in user generated formatting. The default value on single number input is -1.

DR0
DR1 --

Installation dependent commands to select disc drives, by presetting OFFSET. The contents of OFFSET is added to the block number in BLOCK to allow for this selection. Offset is supressed for error text so that it may always originate from drive 0.

DROP n -- LO

Drop the number from the stack.

DUP n -- n n LO

Duplicate the value on the stack.

ELSE addr1 n1 -- addr2 n2 (compiling)
P,C2,LO

Occurs within a colon-definition in the form:

```
IF  ...  ELSE  ...  ENDIF
```

At run-time, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after the ENDIF. It has no stack effect.

At compile-time, ELSE emplaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error testing. ELSE also resolves the pending forward branch from IF by calculating the offset from addr1 to HERE and storing at addr1.

EMIT c -- LO

Transmit ASCII character c to the selected output device. OUT is incremented for each character output.

EMPTY-BUFFERS LO

Mark all block-buffers as empty, not necessarily affecting the contents. Up-dated blocks are not written to the disc. This is also an initialization procedure before first use of the disc. Alias is MTB.

ENCLOSE addr1 c -- addr1 n1 n2 n3

The text scanning primitive used by WORD. From the text address addr1 and an ASCII delimiting character c, is determined the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This procedure will not process past an ASCII "null", treating it as an unconditional delimiter.

END P,C2,LO

This is an "alias" or duplicate definition for UNTIL.

ENDIF addr n -- (compile) P,CO,LO

Occurs in a colon-definition in form:

```
IF ... ENDIF
IF ... ELSE ... ENDIF
```

At run-time, ENDIF serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure. THEN is another name for ENDIF. Both names are supported in fig-FORTH. See also, IF and ELSE.

At compile-time, ENDIF computes the forward branch offset from addr to HERE and stores it at addr. n is used for error tests.

ERASE addr n --

Clear a region of memory to zero from addr over n addresses.

ERROR n -- in blk

Execute error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to screen 176 of drive 0 is printed. This line number may be positive or negative, and beyond just screen 176. If WARNING=0, n is just printed as a message number (non disc installation). If WARNING is -1, the definition (ABORT) is executed, which executes the system ABORT. The user may cautiously modify this execution by altering (ABORT). fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT.

EXECUTE addr --

Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.

EXPECT addr count -- LO

Transfer characters from the terminal to address, until a "return" or the count of characters have been received. One or more nulls are added at the end of the text.

FENCE -- addr U

A user variable containing an address below which FORGETting is trapped. To forget below this point, the user must alter the contents of FENCE.

FILL addr quan b --

Fill memory at the address with the specified quantity of bytes b.

FIRST -- addr

A constant that leaves the address of the first (lowest) block buffer.

FLD -- addr U

A user variable for control of number output field width. Presently unused in fig-FORTH.

FORGET -- E,LO

Executed in the form:

```
FORGET cccc
```

Delete definition named cccc from the dictionary with all entries physically following it.

FORTH -- P,L1

The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. Until additional user vocabularies are defined, new user definitions become a part of FORTH. FORTH is immediate, so it will execute during the creation of a colon-definition, to select this vocabulary at compile time.

FULLK -- V

Sets C/L to 64 and B/SCR to 8, producing 1024 byte screen operation. May be SAVEd in this condition. (See HALFK)

GFLAG -- addr V

A variable that holds a Graphics mode cursor control flag. When the value at GFLAG is non-zero, valForth assumes a split-screen is operative, and will use the alternate cursor-address variables provided by the Operating System to use the text window at the bottom of the display.

HALFK -- V

Sets C/L to 32 and B/SCR to 4, producing 512 byte screen operation. May be SAVEd in this condition. (See FULLK)

HERE -- addr LO

Leave the address of the next available dictionary location.

HEX -- LO

Set the numeric conversion base to sixteen (hexadecimal).

HLD -- addr LO

A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c -- LO

Used between <# and #> to insert an ASCII character into a pictured numeric output string.
e.g. 2E HOLD will place a decimal point.

I -- n C,LO

Used within a DO-LOOP to copy the loop index to the stack. Other use is implementation dependent. See R.

I' -- n B

Copies the second item on the return stack to the stack. Generally used to get the index of the present DO LOOP after an item has been pushed to the return stack for convenience.

ID. addr --

Print a definition's name from its name field address.

IF f -- (run-time)
    -- addr n (compile) P,C2,LO

Occurs in a colon-definition in form:
    IF (tp) ... ENDIF
    IF (tp) ... ELSE (fp) ... ENDIF
At run-time, IF selects execution based on a boolean flag. If f is true (non-zero), execution continues ahead thru the true part. If f is false (zero), execution skips till just after ELSE to execute the false part. After either part, execution resumes after ENDIF. ELSE and its false part are optional.; if missing, false execution skips to just after ENDIF.
At compile-time IF compiles OBRANCH and reserves space for an offset at addr. addr and n are used later for resolution of the offset and error testing.

IMMEDIATE --
Mark the most resently made definition so that when encountered at compile time, it will be executed rather than being compiled, i.e. the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceeding it with [COMPILE].

IN -- addr LO
A user variable containing the byte offset within the current input text buffer (terminal or disc) from which the next text will be accepted. WORD uses and moves the value of IN.

INDEX from to --
Print the first line of each screen over the range from, to. This is used to view the comment (first) lines of an area of text on disc screens.

INTERPRET --
The outer text interpreter which sequentially executes or compiles text from the input stream (terminal or disc) depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT it is converted to a number according to the current base. That also failing, an error message echoing the name with a " ?" will be given. Text input will be taken according to the convention for WORD. If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. See NUMBER.

J -- n B
Copies the third item on the return stack to the stack. Generally used to get the index of the next outer DO LOOP.

KEY -- c LO
Leave the ASCII value of the next terminal key struck.

KLOAD screen -- V
If C/L has a value other than 64, then the number on stack is doubled. In either case, LOAD is then executed. The purpose is to allow smart conditional loading of either 1K screen or 1/2K screen formats. See '(.

LABEL cccc, --
cccc, -- addr

At compilation time, creates a word cccc. At run time, cccc leaves the address of its pfa on the stack. Used to set up a pointer to the following area of memory, as for a machine language subroutine or a player image. Examples:

Example 1: Player image

```
2 BASE !
LABEL UPARROW
00011000  C,
00111100  C,
01111110  C,
00011000  C,
00011000  C,
00011000  C,
DCX
```

Example 2: Machine code two-times

```
ASSEMBLER
LABEL 2*  0 ,X ASL,   1 ,X ROL,   RTS,
```

LATEST -- addr

Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE -- C,LO

Force termination of a DO-LOOP at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA pfa -- lfa

Convert the parameter field address of a dictionary definition to its link field address.

LIMIT -- n

A constant leaving the address just above the highest memory available for a disc buffer.

LIST n -- LO

Display the ASCII text of screen n on the selected output device. SCR contains the screen number during and after this process.

LIT -- n C2,LO

Within a colon-definition, LIT is automatically compiled before each 16 bit literal number encountered in input text. Later execution of LIT causes the contents of the next dictionary address to be pushed to the stack.

LITERAL n -- (compiling) P,C2,LO

If compiling, then compile the stack value n as a 16 bit literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

```
: xxx [calculate] LITERAL ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is resumed and LITERAL compiles this value.

LOAD n -- LO

Begin interpretation of screen n. Loading will terminate at the end of the screen or at ;S. See ;S and -->.

LOOP addr n -- (compiling) P,C2,LO

Occurs in a colon-definition in form:

```
DO ... LOOP
```

At run-time, LOOP selectively controls branching back to the corresponding DO based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead.

At compile-time, LOOP compiles (LOOP) and used addr to calculate an offset to DO. n is used for error testing.

M* n1 n2 -- d

A mixed magnitude math operation which leaves the double number signed product of two signed numbers.

M/ d n1 -- n2 n3

A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.

M/MOD ud1 u2 -- u3 ud4

An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

MAX n1 n2 -- max LO

Leave the greater of two numbers.

MESSAGE n --
Print on the selected output device the text of line n relative to screen 176 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number (disc unavailable).

MIN n1 n2 -- min LO
Leave the smaller of two numbers.

MINUS n1 -- n2 LO
Leave the two's complement of a number.

MOD n1 n2 -- mod LO
Leave the remainder of n1/n2, with the same sign as n1.

MTB -- V
Alias of EMPTY-BUFFERS.

NEXT -- addr
This is the inner interpreter that uses the interpretive pointer IP to execute compiled Forth definitions. It is not directly executed, but is the return point for all code procedures. It acts by fetching the address pointed by IP, storing this value in register W. It then jumps to the address pointed to by the address pointed to by W. W points to the code field of a definition which contains the address of the code which executes for that definition. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of Forth. (Assembler Vocabulary)

NFA pfa -- nfa
Convert the parameter field address of a definition to its name field.

NOOP -- V
A word that does nothing in minimal time. May be used for reserving space in a definition or as a null operation for a word that uses @EX. Generally for advanced programmers. Identical to TASK. Pronounced "no-op".

NOT n -- flag V
Leaves a true flag if n is equal to 0. Otherwise, leaves a false flag.

NUMBER addr -- d
Convert a character string left at addr with a preceeding count, to a signed double number, using the current numeric base. If a decimal point is encountered in the text, its position will be given in DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

O+S start count -- upper-limit+1 lower-limit V
Same as OVER + SWAP. Used to set up limits for DO LOOPs and the like.

OFF -- 0 V
A CONSTANT equal to 0. Used to enhance readability.

OFFSET -- addr U

A user variable which may contain a block offset to disc drives. The contents of OFFSET is added to the stack number by BLOCK. Messages by MESSAGE are independent of OFFSET. See BLOCK, DR0, DR1, MESSAGE.

ON -- 1 V

A CONSTANT equal to 1. Used to enhance readability.

OR n1 n2 -- or LO

Leave the bit-wise logical or of two 16 bit values.

OUT -- addr U

A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formatting.

OVER n1 n2 -- n1 n2 n1 LO

Copy the second stack value, placing it as the new top.

PAD -- addr LO

Leave the address of the text output buffer, which is a fixed offset above HERE.

PFA nfa -- pfa

Convert the name field address of a compiled definition to its parameter field address.

PFLAG -- addr V

A variable that holds an output-select value. If bit 0 is set then output will be sent to the display screen. If bit 1 is set, then output will be sent to the printer. If both bits are set, then output will go to both channels.

PICK ... n -- ... n1 V

Copies the nth entry below n on stack to top of stack. 2 PICK is the same as OVER, 1 PICK is the same as DUP.

POP -- addr

The code sequence to remove a stack value and return to NEXT. POP is not directly executable, but is a Forth re-entry point after machine code. (Assembler Vocabulary)

PREV -- addr

A variable containing the address of the disc buffer most recently referenced. The UPDATE command marks this buffer to be later written to disc.

PROMPT -- V

Intended for system use only, in QUIT. A smart version of the usual ." ok" in QUIT. Prevents "ok" and visible stack printout from being routed to printer.

PUSH -- addr

This code sequence pushes machine registers to the computation stack and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code. (Assembler Vocabulary)

PUT -- addr

This code sequence stores machine register contents over the topmost computation stack value and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code.

QUERY --

Input 80 characters of text (or until a "return") from the operators terminal. Text is positioned at the address contained in TIB with IN set to zero.

QUIT -- L1

Clear the return stack, stop compilation, and return control to the operators terminal. No message is given.

R -- n

Copy the top of the return stack to the computation stack.

R# -- addr U

A user variable which may contain the location of an editing cursor, or other file related function.

R/W addr blk f --

The fig-FORTH standard disc read-write linkage. addr specified is the source or destination block buffer, blk is the sequential number of the referenced block; and f is a flag for f=0 write and f=1 read. R/W determines the location on mass storage, performs the read-write and performs any error checking. Important: See Note 1 at end of glossary.

R> -- n LO

Remove the top value from the return stack and leave it on the computation stack. See >R and R.

R0 -- addr U

A user variable containing the initial location of the return stack. Pronounced R-zero. See RP!

REPEAT addr n -- (compiling) P,C2

Used within a colon-definition in the form:

BEGIN ... WHILE ... REPEAT

At run-time, REPEAT forces an unconditional branch back to just after the corresponding BEGIN.

At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr. n is used for error testing.

ROLL ... n -- ... V

Moves the nth entry below n on stack to top of stack. 3 ROLL is the same as ROT, 2 ROLL is the same as SWAP. 0 ROLL is undefined.

ROT n1 n2 n3 --- n2 n3 n1 LO

Rotate the top three values on the stack, bringing the third to the top.

RP! --

A computer dependent procedure to initialize the return stack pointer from user variable R0.

RPICK n -- V
Copies nth entry on return stack to parameter (number) stack. For instance, 1 RPICK is the same as R and 2 RPICK is the same as I', etc.

S->D n -- d
Sign extend a single number to form a double number.

SAVE -- V
Gives prompt, and if answered with press of "Y" key, moves COLD and FENCE parameters to cover current system, makes Forth current, and creates a bootable copy of RAM-resident system up to HERE on disk in drive 1.

S0 -- addr U
A user variable that contains the initial value for the stack pointer. Pronounced S-zero. See SP!

SCR -- addr U
A user variable containing the screen number most recently referenced by LIST.

SGRCTL -- addr V
"Shadow Register" for GRACTL, the Atari graphics control register. See Atari Operating System Manual for explanation.

SIGN n d -- d LO
Stores an ASCII "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is maintained. Must be used between <# and #>.

SMUDGE --
Used during word definition to toggle the "smudge bit" in a definitions' name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SP! --
A computer dependent procedure to initialize the stack pointer from S0.

SP@ -- addr
A computer dependent procedure to return the address of the stack position to the top of the stack, as it was before SP@ was executed. (e.g. 1 2 SP@ @ . . . would type 2 2 1)

SPACE -- LO
Transmit an ASCII blank to the output device.

SPACES n -- LO
Transmit n ASCII blanks to the output device.

SPEMIT c -- V
Like EMIT, but defined for Atari. Will output control codes as characters instead of executing the controls. Used by EXPECT and other words.

STATE -- addr LO,U

A user variable containing the compilation state. A non-zero value indicates compilation. The value itself may be implementation dependent.

SWAP n1 n2 -- n2 n1 LO

Exchange the top two values on the stack.

TASK --

A no-operation word which can mark the boundary between applications. By forgetting TASK and re-compiling, an application can be discarded in its entirety.

THEN P,CO,LO

An alias for ENDIF.

TIB -- addr U

A user variable containing the address of the terminal input buffer.

TOGGLE addr b --

Complement the contents of addr by the bit battern b.

TRAVERSE addr1 n -- addr2

Move across the name field of a fig-FORTH variable length name field. addr1 is the address of either the length byte or the last letter. If n=1, the motion is toward hi memory; if n=-1, the motion is toward low memory. The addr2 resulting is address of the other end of the name.

TRIAD scr --

Display on the selected output device the three screens which include that numbered scr, beginning with a screen evenly divisible by three. Output is suitable for source text records, and includes a reference line at the bottom taken from line 14 of screen 177.

TYPE addr count -- LO

Transmit count characters from addr to the selected output device.

TYPE as supplied zeroes out the high bit of each character before sending it to the output device, usually the screen or printer. If you want to be able to type all 8 bits for inverse characters, do the following:

```
255   ' TYPE  20 + C!
```

and return to 7 bit output by doing

```
127   ' TYPE 20  + C!
```

More generally,

```
: 78TYPE  < ' TYPE 14 + >  C! ;
: 7TYPE   127 78TYPE ;
: 8TYPE   255 78TYPE ;
```

(What you are doing with all of this is changing the mask that TYPE uses before executing EMIT.)

U* u1 u2 -- ud

Leave the unsigned double number product of two unsigned numbers.

U. n -- B

Prints the number n in unsigned form.

U.R u n -- B

Prints unsigned number u right justified in a field n wide.

U/ ud u1 -- u2 u3

Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

U> u2 u1 -- flag V

Leaves true flag is u2 (unsigned) is greater than u1 (unsigned). Otherwise, leaves false flag.

U? addr -- V

Does a @ from addr and an unsigned print of top of stack.

UNTIL f -- (run-time)
addr n -- (compile) P,C2,LO

Occurs within a colon-definition in the form:

```
BEGIN ... UNTIL
```

At run-time, UNTIL controls the conditional branch back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN; if true, execution continues ahead.

At compile-time, UNTIL compiles (OBRANCH) and an offset from HERE to addr. n is used for error tests.

UPDATE -- LO

Marks the most recently referenced block (pointed to by PREV) as altered. The block will subsequently be transferred automatically to disc should its buffer be required for storage of a different block.

USE -- addr

A variable containing the address of the block buffer to use next, as the least recently written.

USER n -- LO

A defining word used in the form:

```
n USER cccc
```

which creates a user variable cccc. The parameter field of cccc contains n as a fixed offset relative to the user pointer register UP for this user variable. When cccc is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable.

VARIABLE n -- E,LU

A defining work used in the form:

```
n VARIABLE cccc
```

When VARIABLE is executed, it creates the definition cccc with its parameter field initialized to n. When cccc is later executed, the address of its parameter field (containing n) is left on the stack, so that a fetch or store may access this location.

VOC-LINK -- addr U

A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting thru multiple vocabularies.

VOCABULARY -- E,L

A defining word used in the form:

```
VOCABULARY cccc
```

to create a vocabulary definition cccc. Subsequent use of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the CURRENT vocabulary into which new definitions are placed.

In fig-FORTH, cccc will be so chained as to include all definitions of the vocabulary in which cccc is itself defined. All vocabularies ultimately chain to Forth. By convention, vocabulary names are to be declared IMMEDIATE. See VOC-LINK.

VLIST --

List the names of the definitions in the context vocabulary.

WAIT -- V

Halts execution of Forth until a CONSOLE button is pressed.

WARNING -- addr U

A user variable containing a value controlling messages. If = 1 disc is present, and screen 176 of drive 0 is the base location for messages. If = 0, no disc is present and messages will be presented by number. If = -1, execute (ABORT) for a user specified procedure. See MESSAGE, ERROR.

WHILE f -- (run-time)
adl nl -- adl nl ad2 n2 P,C2

Occurs in a colon-definition in the form:

BEGIN ... WHILE (tp) ... REPEAT

At run-time, WHILE selects conditional execution based on boolean flag f. If f is true (non-zero), WHILE continues execution of the true part thru to REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure.

At compile time, WHILE emplaces (OBRANCH) and leaves ad2 of the reserved offset. The stack value will be resolved by REPEAT.

WIDTH -- addr U

In fig-FORTH, a user variable containing the maximum number of letters saved in the compilation of a definitions' name. It must be 1 thru 31, with a default value of 31. The name character count and its natural characters are saved, up to the value in WIDTH. The value may be changed at any time within the above limits.

WORD c -- LO

Read the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurances of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disc block stored in BLK. See BLK, IN.

X --

This is pseudonym for the "null" or dictionary entry for a name of one character of ASCII null. It is the execution procedure to terminate interpretation of a line of text from the terminal or within a disc buffer, as both buffers always have a null at the end.

XOR n1 n2 -- xor L1

Leave the bit-wise logical exclusive-or of two values.

ok -- V

Does three backspaces. When using "logical line input" from keyboard to re-input previously entered line, this word gives harmless meaning to the old "ok" prompt Forth may find in the input stream. See "logical line input" section in Strolling through ValForth.

[-- P,L1

Used in a colon-definition in form:

: xxx [words] more ;

Suspend compilation. The words after [are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with]. See LITERAL,].

[Compile] -- P,C

Used in a colon-definition in form:

: xxx [COMPILE] FORTH ;

[COMPILE] will force the compilation of an immediate definition, that would otherwise execute during compilation. The above example will select the FORTH vocabulary when xxx executes, rather than at compile time.

] -- L1

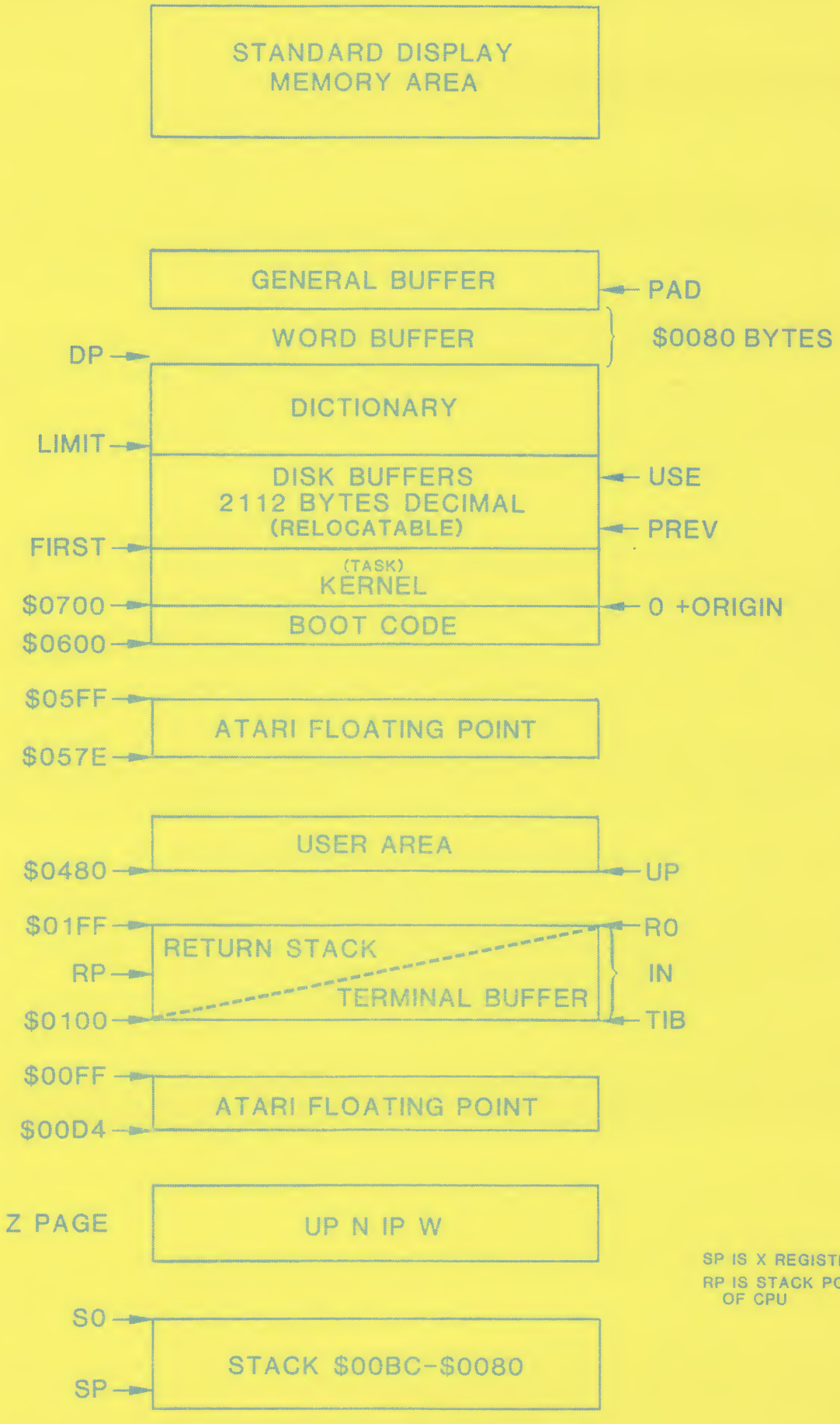
Resume compilation, to the completion of a colon-definition. See [.

NOTES:

Note 1

Due to a bug in at least some of the Atari Operating System ROM programs, a sector may not be written directly from a memory area in which the low byte of the bottom location is $7F. The system will hang if this is attempted. This is not a valFORTH bug, it is Atari's. Please watch out for it.

valFORTH T.M. Memory Map



valFORTH T.M.

Advanced 6502 Macro Assembler

Version 2.0
April 1982

Although the FORTH language is many times faster than BASIC or PASCAL, there are still times when speed is so critical that one must turn to assembly language programming as a matter of necessity. Not wanting to give up the advantages of the FORTH language, FORTH programmers typically use an assembler designed specifically for the FORTH system. valFORTH incorporates a very powerful FORTH style 6502 assembler for these special programming jobs.

Overview

Most programming applications can be undertaken completely in high level FORTH. There are times, due to speed constraints, when assembly language must be used. Typically, "number crunching" and high speed graphic routines must be machine coded. valFORTH provides a powerful 6502 FORTH assembler for these special occasions.

FORTH assemblers differ from standard assemblers by making the best use of the stack and the FORTH system as a whole. The FORTH assembler is smaller than a standard assembler. In the case of the valFORTH assembler, this is particularly true.

The valFORTH assembler offers the programmer the following improvements over a standard assembler:

1) IF...THEN...ELSE structures which use positive logic rather than negative logic.

2) BEGIN...UNTIL structures for post-testing indefinite loops.

3) WHILE...REPEAT structures for pre-testing indefinite loops.

4) BEGIN...AGAIN structures for unconditional looping.

5) Full access to the FORTH operating system and its capabilities such as changing bases.

6) Complex assembly time calculations.

7) Mixed high level FORTH with assembly code to take full advantage of each.

8) Full macro capability.

The following is a complete description of the valFORTH assembler. This description assumes a working knowledge of 6502 assembly language programming and related terms.

The purpose of the FORTH assembler is to allow machine language programming without the need to abandon the FORTH system. Words coded in assembly language must follow the standard FORTH dictionary format and must adhere to certain guidelines regarding their coding.

Assembly language programmers typically have two methods of storing programs into RAM. The machine code can be poked directly into memory, or an assembler can be used to accomplish this. The former method is brutal, but it has the advantage that precious memory is not taken up by the assembler. The drawback, of course, is loss of readability and ease of modification. FORTH allows both of these methods to be employed.

The words "," and "C," can be used to poke any machine language program into the dictionary. This is used only when memory restrictions prohibit the use of an assembler or if it is assumed that no assembler is available.

In high level FORTH, words are compiled into the dictionary using the following form:

```
: name     high-level-FORTH... ;
```

When compiling a machine coded word, this becomes:

```
CODE  name     machine-code... C;
```

In this example, the word "CODE" creates a header for the next word in much the same way ":" creates a header. The difference lies in the fact that ":" informs the system that the following definition is high level FORTH, while "CODE" indicates that the definition is a machine or assembly language definition. In the same manner, ";" terminates a high level FORTH definition while "C;" terminates a code definition.

To clarify this, a code definition will be programmed that will clear the top line of the current video display on an Atari 800 microcomputer. Note that video memory is pointed to by the address stored in locations 88 and 89 (decimal). The 6502 code is shown in listing 1.

```
CLR   TYA           ; Y comes in with 0; 0 means a blank
      LDY #39       ; 40 characters/line (0 thru 39)
LOOP  STA (88), Y   ; Fill from end to beginning
      DEY           ; Done?
      BPL LOOP      ; Keep going if not
      JMP NEXT      ; Re-enter the FORTH operating system
```

Listing 1

The CODE definition equivalent to listing 1 would be:

```
      HEX                                  (put in hex mode)
      CODE CLR                             (define code word)
           98 C,                           (poke in code)
           A0 C, 27 C,
           91 C, 58 C,
           88 C,
           10 C, FB C,
      C; DECIMAL                           (end assembly)
```

First, the FORTH system is put into the hexadecimal mode so that opcode values need not be converted to decimal. Next, the word CODE puts the system into an assembly mode and enters the new word CLR into the dictionary as a machine language word. The opcodes are then byte compiled ("C,") into the dictionary. Note that for the final jump to re-enter FORTH, the predefined word NEXT was word compiled (",") into the dictionary. The word C; terminates the assembly process. The system is then restored to the decimal mode.

This method can always be used, but it is very tedious. Each opcode must be looked up, and all relative branches calculated. Besides introducing a great source for error, if a single opcode is added or deleted, it is possible that many jumps must be re-calculated. For this reason, using the assembler is the prescribed method for entering machine language routines.

Unlike the standard assembler which has four fields (the label field, the operation field, the operand field, and the comment field), the FORTH assembler has only three fields. In a FORTH assembler, there is no explicit label field, but there is an implied label field through the use of the assembler constructs IF, and BEGIN, described later. In addition, the remaining three fields in the FORTH assembler are in reversed order (as is standard for the FORTH language). In other words, the operand precedes the operation, and remarks can be embedded anywhere.

In compiling an assembly word, the FORTH assembler ultimately uses either "," or "C," and for this reason assembly mnemonics traditionally end with a comma. valFORTH equivalents are shown in chart 1.

Standard Assembler	valFORTH Assembler
LDX COUNT	COUNT LDX,
JMP COUNT+1	COUNT 1+ JMP,
LDA #3	# 3 LDA,
ADC N	N ADC,
STY TOP,X	TOP ,X STY,
INC BOT,Y	BOT ,Y INC,
STA (TOP,X)	TOP X) STA,
AND (BOT),Y	BOT)Y AND,
JMP (POINT)	POINT)JMP,
DEC N+4	N 4 + DEC,
DEX	DEX,
ROL A	.A ROL,
	or ROL.A,

Note: # 9 LDA, = 9 # LDA,
TOP ,X ROL, = ,X TOP ROL, etc.

Chart 1

Converting the program given in listing 1 to FORTH assembly mnemonics we have:

```
DECIMAL
CODE CLR
     TYA,                    (    TYA          )
     # 39 LDY,               (    LDY #39      )
     BEGIN,                  ( LOOP            )
        88 )Y STA,           (    STA (88),Y   )
        DEY,                 (    DEY          )
     MI UNTIL,               (    BPL LOOP     )
     NEXT JMP,               (    JMP NEXT     )
C;
```

In the above example, a BEGIN ... UNTIL, clause (described in the next section) is used. By using this structure, no labels are necessary and positive logic is used rather than negative logic (i.e., "repeat until minus" instead of "if NOT minus, then repeat"). Note that the FORTH assembler compiles exactly the same machine code as the standard assembler, it simply makes the assembly coding easier.

Control Structures

Allowing labels within assembly language programming would make the FORTH assembler needlessly long and slow. To get around the problem of test branching, the ValFORTH assembler has a very powerful set of control structures similar to those found in high level FORTH.

The IF,...ENDIF, and IF,...ELSE,...ENDIF, clauses

The IF, construct which handles conditional downward branches has the following two forms:

```
...code...                    ...code...
flag IF,                      flag IF,
    ...true code...               ...true code...
ENDIF,                        ELSE,
...code...                        ...false code...
                              ENDIF,
                              ...code...
```

where "flag" is one of the 6502 statuses: NE , EQ , CC , CS , VC , VS , MI , or PL. The following are a few examples of how these are used.

Note: When the FORTH inner interpreter passes control to an assembly language routine, the Y register always contains a zero value and the X register must be preserved as it is used by the FORTH system to maintain the parameter stack. See the section on parameter passing for more information.

```
;  Code routine for 1+
ONEPL INC 0,X       ; increment low byte of 16 bit value
      BNE THERE     ; carry out of low?
      INC 1,X       ; increment high byte if so
THERE JMP NEXT      ; re-enter FORTH system
```

Now in ValFORTH assembly language:

```
CODE ONEPL          (define word)
     0 ,X INC,      (increment low byte)
     EQ IF,         (if result was zero,)
        1 ,X INC,   (then bump the high byte)
     ENDIF,
     NEXT JMP,      (exit to FORTH)
C;
```

Note: In the following example, CONIN is assumed to be predefined.

```
     ; Input routine
     INPUT JSR CONIN      ; Go get character, comes back in A
           CMP #$OD       ; Is it a carriage return?
           BNE INP1       ; If not, do something else
           ...code1...    ; execute code for carriage return
           JMP INP2       ; do not execute "normal" code
     INP1  ...code2...    ; execute code for normal keys
     INP2  ...code3...    ; execute code more common code
           JMP NEXT       ; re-enter FORTH system
```

The equivalent valFORTH version would be:

```
     HEX
     CODE INPUT
          CONIN JSR              (Get character          )
          # OD CMP,              (carriage return?       )
          EQ IF,                 (If so, then            )
             ...code1...         (execute c/r code       )
          ELSE,                  (otherwise              )
             ...code2...         (execute normal code    )
          ENDIF,
          ...code3...
          NEXT JMP,              (re-enter FORTH system )
     C; DECIMAL
```

The BEGIN,...UNTIL, clause

Another useful structure is the BEGIN,...UNTIL, construct which allows for post-testing indefinite looping. The BEGIN,...UNTIL, construct has the following form:

```
     ...code1...
     BEGIN,                      code2 is repeatedly
        ...code2...              executed until "flag"
     flag UNTIL,                 is true.
     ...code3...
```

The following 6502 routine waits until a carriage return has been typed.

```
     ; WAIT until c/r
     WAIT  JSR CONIN      ; Go get a character, comes back in A
           CMP #$OD       ; Is it a carriage return?
           BNE WAIT       ; Ask again if not
           JMP NEXT       ; Return to FORTH
```

Using the BEGIN, clause, this becomes

```
     NEXT
     CODE  WAIT                   (Code name WAIT  )
           BEGIN,                 (Begin waiting   )
              CONIN JSR,          (Get a character )
              # OD CMP,           (Carriage return?)
           EQ UNTIL,              (loop up until so)
           NEXT JMP,
     C; DECIMAL
```

The BEGIN,...WHILE,...REPEAT, clause

In the valFORTH assembler, there is another valuable control structure. It is the BEGIN,...WHILE,...REPEAT, structure. The WHILE, clause allows pre-testing indefinite loops to be easily programmed. It takes the form:

```
...code1...
BEGIN,                      Code2 and code3 are repeatedly
    ...code2...             executed until "flag" become
flag WHILE,                 false, at which time program
    ...code3...             control proceeds to code4.
REPEAT,
...code4...
```

A common example of the WHILE, clause is getting a line of input text terminated by a carriage return.

```
;  Get line of text  (note: Y=0 on entry always)
GETLN JSR CONIN          ; Get one character
      CMP #$OD           ; C/R terminates input
      BEQ GETL1          ; If not a C/R then
      STA BFFR,Y         ; store the character
      INY                ; Bump buffer pointer
      JMP GETLN          ; Go back for more
GETL1 JMP NEXT           ; Exit to FORTH
```

Using the WHILE, clause in valFORTH, we have:

```
HEX
CODE GETLN
     BEGIN,
        CONIN JSR,        (Get a character          )
        # OD CMP,         (Carriage return?        )
     NE WHILE,           (If not,                 )
        BFFR ,Y STA,      (then store the character )
        INY,              (and bump the pointer    )
     REPEAT,              (Repeat all of the above )
     NEXT JMP,
C; DECIMAL
```

The BEGIN,...AGAIN, clause

The final control structure is the BEGIN,...AGAIN, structure. This structure allows the use of unconditional looping in assembly language routines. Although its use is rare, it can reduce code size considerably. It takes the following form:

```
...code1...
BEGIN,                              Repeatedly execute code2
   ...code2...                      and code4 until "flag"
   flag IF,                         becomes true, in which
      ...code3...                   case, program execution
      re-entry-point JMP,           continues with code3 and
   ENDIF,                           a system re-entry made.
   ...code4...
AGAIN, C;
```

The best example of the AGAIN, clause is in the coding of the CMOVE routine:

```
;  Byte at a time front end memory move
CMOVE LDA #3              ; Get top three stack items
      JSR SETUP           ; Move them to N scratch area
CMOV1 CPY N               ; Time to decrement COUNT high?
      BNE CMOV2           ; Nope
      DEC N+1             ; Yes, so do it
      BPL CMOV2           ; Bypass exit if not done
      JMP NEXT            ; Exit to FORTH system
CMOV2 LDA (N+4),Y         ; Get byte to move
      STA (N+2),Y         ; Move it!
      INY                 ; Bump byte pointer
      BNE CMOV1           ; Keep going until ready to
      INC N+5             ;   bump high bytes of both
      INC N+3             ;   "to" and "from" addresses
      JMP CMOV1           ; Do it all again
```

Using the AGAIN, clause, this becomes:

```
CODE CMOVE
     # 3 LDA,                    (Prepare for memory move)
     SETUP JSR,
     BEGIN,
        BEGIN,                  (Start the process)
           N CPY,               (done?)
           EQ IF,
              N 1+ DEC,         (Maybe, keep checking)
              MI IF,
                 NEXT JMP,      (Re-enter FORTH system)
              ENDIF,
           ENDIF,
           N 4 + )Y LDA,        (Get byte to copy)
           N 2+ )Y STA,         (Store in new location)
           INY,                 (Bump pointer)
        EQ UNTIL,
        N 5 + INC,              (Bump addresses)
        N 3 + INC,
     AGAIN,                     (Do it all again)
C; DECIMAL
```

Parameter Passing

One of the most useful features of the FORTH language is its ability to use a parameter stack for passing values from one word to another. For assembly language routines to really be useful in the FORTH system, there must be some facility for these routines to access this stack. Likewise, there should be some way in which to access the return stack as well. This section details exactly how to make the best use of both stacks.

Since the FORTH system maintains dual stacks and the 6502 supports only one, it is necessary to simulate one of the stacks. For ease of stack manipulation, the parameter is simulated; the return stack uses the hardware stack of the microprocessor.

The simulated stack uses the 0-page,X addressing mode of the 6502. For example, the following statements show how the parameter stack is organized.

```
LDA 0,X      Low byte of item on top of stack
INC 1,X      High byte of top item
ADC 2,X      Low byte of item second on stack
EOR 3,X      High byte of 20S
RNL 4,X      Low byte of item third on stack
AND 5,X
...          etc.
```

In high level FORTH, the word DROP drops (or pops) the top value from the stack. The code definition for DROP is:

```
CODE DROP     INX, INX, NEXT JMP, C;
```

In the same way, values can be "pushed" to the stack. Note that the X register must be preserved between FORTH words or the parameter stack is lost! Thus if the X register is needed in a code definition, it must be saved upon entry to the routine and restored before returning to the FORTH system. The special location XSAVE is reserved for this: (The word XSAVE has been defined as a FORTH constant.)

```
STX XSAVE        Save the X register
LDX XSAVE        Restore the X register
```

In all the examples given so far, the code definitions have re-entered the FORTH system through the normal re-entry point called NEXT. The following is a complete description of all possible re-entry points: (In all of the following code examples, standard 6502 assembler format has been used for ease of comprehension. All valFORTH assembler equivalents can be found in appendix A.)

The NEXT re-entry point

The NEXT routine transfers control to the next FORTH word to be executed. All FORTH words eventually come through the NEXT routine. Likewise, all other re-entry points come through NEXT once they have completed their special tasks. The next routine is typically used by words

The NEXT re-entry point (cont'd)

such as 1- which do not modify the number of arguments on the stack. The word NEXT is defined as a FORTH constant. NXT, is an abbreviation for NEXT JMP, .

Example:

```
; 1- routine
ONEM LDA 0,X      ; Borrow from low byte?
     BNE ONE1     ; If not, ignore correction
     DEC 1,X      ; Decrement high byte
ONE1 DEC 0,X      ; Now do the low
     JMP NEXT     ; Re-enter FORTH
```

Listing 2

The PUSH re-entry point:

The PUSH routine pushes a 16 bit value to the parameter stack whose low byte is found on the 6502 return stack and whose high byte is found in the accummulator. The X register is automatically decremented twice for the two bytes. This routine is typically used for words such as OVER or DUP which leave one more argument than they expect. The word PUSH has been defined as a FORTH constant. PSH, is an abbreviation for PUSH JMP, .

Example:

```
; DUP routine
DUP  LDA 0,X      ; Get low byte of TOS
     PHA          ; Push it
     LDA 1,X      ; Put high byte in A
     JMP PUSH     ; Put it on the P-stack
```

Listing 3

The PUT re-entry point:

The PUT routine replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is found on the 6502 stack and whose high byte is in the accumulator. This is used by words such as ROT or SWAP which do not change the number of values on the stack. The word PUT has been defined as a FORTH constant. PUT, is an abbreviation for PUT JMP, .

Example:

```
; SWAP routine
SWAP LDA 2,X      ; Low byte of 2nd value
     PHA          ; Save it
     LDA 0,X      ; Put low byte of TOS
     STA 2,X      ; into low byte of 2OS
     LDA 3,X      ; Hold high byte of 2OS
     LDY 1,X      ; Put high byte of TOS
     STY 3,X      ; into high byte of 2OS
     JMP PUT      ; Replace TOS no
```

Listing 4

The PUSHOA re-entry point

The PUSHOA re-entry point pushes the 8 bit unsigned value in the accumulator as a 16 bit value with the upper 8 bits zeroed. This word is very commonly used by words which leave a boolean flag on the parameter stack such as ?TERMINAL. The word PUSHOA has been defined as a FORTH constant. PSHA, is an abbreviation for PUSHOA JMP, .

Example:
```
;  ?TERMINAL routine
QTERM LDA $D01F     ; Read Atari CONSOLE keys
      EOR #7        ; Anything pressed?
      BEQ QT1       ; If not, go push false
      INY           ; Else push a true
QT1   TYA           ; Put Y (0 or 1) in A
      JMP PUSHOA    ; Go push the result
```

Listing 5

The PUTOA re-entry point:

The PUTOA routine replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is in the accumulator and whose high byte is set to zero. This is used by words such as C@ which simply replace their arguments on the stack. The word PUTOA is defined as a FORTH constant. PUTA, is an abbreviation for PUTOA JMP, .

Example:
```
;  Byte fetch
CFCH LDA (0,X)      ; Load byte indirectly
     JMP PUTOA      ; Replace the address
                    ; with the contents
```

Listing 6

The BINARY re-entry point

The BINARY re-entry point drops the value on top of the parameter stack and then performs the PUT operation described above. This word is commonly used by words such as XOR which use one more argument than they leave. The word BINARY has been defined as a FORTH constant.

Example:
```
;  Exclusive or TOS with 2OS
XOR   LDA 0,X       ; Get low byte of top value
      EOR 2,X       ; XOR it with low of 2OS
      PHA           ; Save it
      LDA 1,X       ; Now do same for high bytes
      EOR 3,X       ; Result in A
      JMP BINARY    ; Go DROP , PUT
```

Listing 7

POP and POPTWO re-entry points

The POP and POPTWO re-entry points are used when values must be dropped from the parameter stack. POP performs a DROP, while POPTWO performs a 2DROP. Most words which can use BINARY can use POP. The words POP and POPTWO have been defined as FORTH constants. POP, is an abbreviation for POP JMP, and POP2, is an abbreviation for POPTWO JMP, .

Examples:

```
;  Another XOR routine
XOR  LDA 0,X        ; Get low byte
     EOR 2,X        ; XOR with other low byte
     STA 2,X        ; Put directly on stack
     LDA 1,X        ; Do the same for high bytes
     EOR 3,X
     STA 3,X
     JMP POP        ; Remove unneeded TOS item
```

Listing 8

```
;  C! routine
CSTR LDA 2,X        ; Get byte to store
     STA (0,X)      ; Store it!
     JMP POPTWO     ; Drop byte and address
```

Listing 9

The SETUP routine

A very useful routine in the FORTH system is the code routine SETUP. On the 6502, 0-page addressing is typically faster than absolute addressing. Also, some instructions, such as indirect-indexed addressing, can use only 0-page addresses. The SETUP routine allows the assembly language programmer to transfer up to four stack values to a scratch pad in the 0-page for these operations. The predefined name for this area is N. The calling sequence for the SETUP routine is:

```
LDA #num      ; Move "num" values to N, ("num" = 1-4)
JSR SETUP     ; then drop "num" values from the stack
```

The SETUP routine moves one to four values to the N scratch area and drops all values moved from the parameter stack. These values are stored in the following order:

```
LDA N       ; Low byte of value that was TOS
EOR N+1     ; High byte            ( N 1+ EOR, )
ADC N+2     ; Low byte of value that was 2OS
STY N+3     ; High byte            ( N 3 + STY, )
INC N+4     ; Low byte of 3OS      ( N 4 + INC, )
...
DEC N+7     ; High byte of value that was 4OS
```

Words such as CMOVE and FILL which use indirect-indexed addressing typically use the SETUP routine (see the BEGIN,...AGAIN, example). The word SETUP has been defined as a FORTH constant.

Return stack manipulation

The FORTH return stack is implemented as the normal 6502 hardware stack. To push and pop values, the 6502 stack instructions PHA and PLA can be used. Sometimes it is also necessary to manipulate the data on the return stack (such as for DO looping). Using the normal stack operations to do this can be tedious. Using indexed addressing, the return stack can be manipulated in the same manner as the parameter stack.

Examples:

```
;  >R routine
TOR  LDA 1,X        ; Pick up high byte
     PHA            ; Push it to R
     LDA 0,X        ; Now do the low byte
     PHA            ; It's done!
     JMP POP        ; Now, "lose" TOS
```

Listing 10

```
;  3rd loop index (I , I' , J , ... K )
K    STX XSAVE      ; Save P-stack pointer
     TSX            ; Get R-stack pointer
     LDA $109,X     ; 101-102,...,109-10A, (L-H)
     PHA            ; Push low byte of 3rd item
     LDA $10A,X     ; A now has high byte
     LDX XSAVE      ; Restore P-stack pointer
     JMP PUSH       ; Push the index
```

Listing 11

Machine Language Subroutines in valFORTH

When coding in assembly language, it is often useful to be able to make subroutine calls for often used operations. Using CODE makes it possible to do this, but it is not recommended. The following subroutine uses CODE.

```
CODE S1+                    ( Subroutine 1+      )
     0 ,X INC,              (    INC 0,X         )
     EQ IF,                 (    BNE *+4         )
       1 ,X INC,            (    INC 1,X         )
     ENDIF,                 (                    )
     RTS,                   (    RTS             )
C;
```

This subroutine could now be used in assembly language routines in the following way:

```
CODE 1+                     (Another 1+ routine  )
     ' S1+ JSR,             (   JSR S1+          )
     NEXT  JMP,             (   JMP NEXT         )
C;
```

This works fine, but there is one slight problem. If the user types S1+ as a command (i.e., it is not called, but executed) the FORTH system will "crash" when the RTS statement is encountered. This is because FORTH does not call its words, but jumps to them. For this reason, CODE is not used. A word which acts like CODE but protects the system is needed.

In the code for 1+ above, it was necessary to ' (tick) the subroutine to find its address. It would be desirable if we could simply type its name and have it return its address (just as NEXT and PUSH do). This is possible. The word SUBROUTINE below allows this (note that this word is not automatically loaded with the assembler, it must be typed in by the user).

```
: SUBROUTINE                    (new word SUBROUTINE     )
  0 VARIABLE                    (is like a VARIABLE      )
  -2 ALLOT                      (discard the value of 0  )
  [COMPILE] ASSEMBLER           (Put into assembly mode  )
  ?EXEC  !CSP ;                 (Set/check for errors    )
```

The word SUBROUTINE can be used in the same way CODE is except that SUBROUTINEs end with an RTS instruction while CODE routines must end with a jump to a re-entry point. When the word defined using SUBROUTINE is executed, the entry point to the routine is left on the stack similar to the way in which a word defined using VARIABLE leaves an address. The following is an example of subroutine usage.

```
SUBROUTINE 2'SCOMP              (Two's complement  )
   SEC,                         (routine           )
   0  # LDA,
   0 ,X SBC,                    (i.e., TOS => - TOS)
   0 ,X STA,
   0  # LDA,
   1 ,X SBC,
   1 ,X STA,
   RTS,
C;
```

It can now be used as such:

```
CODE ABS                        (Take abs. value of TOS )
     1 ,X LDA,                  (Is TOS < 0 ?           )
     MI IF,
        2'SCOMP JSR,            (If so, TOS => -TOS     )
     ENDIF,
     NEXT JMP,                  (Exit to FORTH system   )
```

When the new word 2'SCOMP is executed directly, it leaves its address on the stack. When it is called by a subroutine, it performs a two's complement on the top stack value. This dual type of execution allows safe access to assembly language subroutines.

Macro Assemblies in valFORTH

FORTH assemblers use a reversed form of notation so that all the benefits of the standard FORTH system are available. In other words, anything that can be done in FORTH can be done during assembly time in a code definition. This is because all of the assembler opcodes are actually FORTH words which take arguments from the parameter stack. Thus

```
NEXT JMP,
```

actually puts the address of the NEXT routine (NEXT is a FORTH constant) onto the parameter stack. The word JMP, then compiles the address into the dictionary. Here is a simplified definition (it does not test for indirect jumping) for the JMP, opcode:

```
HEX
: JMP,                  (address --- )
  4C C,                 (compile in JMP opcode )
  ,                     (compile in the address )
; DECIMAL
```

All assembly words are designed in this fashion. Thus the necessity for operands to precede opcodes becomes clear. This allows the use of complex assembly time calculations that no ordinary assembler would ever support (e.g. no standard 6502 assembler would allow the use of the SIN function for generating a data table).

Most assemblers do allow the use of the basic operations: + , - , * , / , and &. These are easily used in the valFORTH system:

```
LDA #COUNT&$FF          COUNT FF AND # LDA,
LDY #NAME/$100          COUNT 100 /  # LDY,
EOR N+6                   N     6 +    EOR,
LDX #"A"+$80            ASCII A 80 + # LDX,
etc.
```

The looping structures IF, and BEGIN, each leave two values on the stack during assembly time. The first is a branch address, the second is an identification code. When ENDIF, is executed, it checks the identification code to verify that structures have not been illegally interleaved (i.e., BEGIN, ... ENDIF,). If everything checks out, ENDIF, then calculates the branch offset required by the IF, clause, otherwise an error is reported. The BEGIN, clause functions in the same manner. Thus, the words IF, and BEGIN, are predefined macro instructions in the valFORTH assembler.

The fact that a FORTH assembler is nothing but a collection of words means that the assembler, like the FORTH language itself, is extensible. In other words, macro assemblies can easily be performed by defining new assembler directives. Take the following code extract which outputs a text string:

```
...code...                  ; ...
JSR CRLF                    ; Skip to next line
JSR PRTXT                   ; Call print routine
.BYTE 11,'valFORTH 1.'      ; String to output
LDA REL                     ; Get release number
JSR PRTNM                   ; Print the number
JSR CRLF                    ; Issue c/r
...code...                  ; ...
```

This code prints out the string "valFORTH 1.x" where "x" is the release number. Note that the routine PRTXT does not exist, it is simply used here for example purposes. The PRTXT routine "pops" the return address which points to the output string, picks up the length byte and adds it to the return address. The return address, which now points to the LDA instruction is "pushed" back onto the stack. The PRTXT routine still has a pointer to the string which it then prints out. Finally, it dones an RTS and returns control to the calling program. The release number is then printed out.

Assuming that the PRTXT routine is used quite often, it would be desirable to make it an assembler macro. A word which automatically assembles in the subroutine call to PRTXT and then assembles in a user specified string would be quite handy. In valFORTH, this is easily accomplished:

```
ASSEMBLER DEFINITIONS       (This is an assembler word)
HEX                         (Put system in base 16)
: PRINT"                    (Command form:  PRINT" text")
  20 C,  PRTXT ,            (compile in JSR PRTXT)
  22 WORD                   (Now the string upto ")
  HERE C@ 1+ ALLOT          (Bump dictionary pointer)
; DECIMAL                   (all done,)
IMMEDIATE                   (make word execute even at)
FORTH DEFINITIONS           (compile time.)
```

This word could now be used in ValFORTH assemblies in the following manner:

```
...code...
CRLF JSR,                   (Skip to next line)
PRINT" ValFORTH 1."         (Print out string)
REL LDA,                    (Get release number)
PRTNM JSR,                  (Go print it)
CRLF JSR,                   (Skip to next line)
...code...
```

Using the newly defined PRINT" macro, strings no longer need to be counted, and since there is less text to enter, typing errors are reduced. Other useful macros which could be designed are words which allow conditional assembly or automatically set up IOCB blocks for Atari operating system calls. Experienced assembly language programmers typically have a set of often used routines defined as macro instructions for quicker program development.

Compatability With Other Popular Assemblers

There are several other versions of FORTH out which have 6502 assemblers. The two major versions are the Forth Interest Group's written by William Ragsdale, and the version put out by the Atari Program Exchange written by Patrick Mullarky. The valFORTH assembler is a superset of both of these fine assemblers and is fully compatible with both versions.

Although not stated previously in the documentation, there are several ways in which to implement the IF, , WHILE, and UNTIL, structures. The valFORTH assembler was designed with transportability in mind. Although the recommended method is the valFORTH version, each of the following may be used.

valFORTH	Fig version	APX version
EQ IF,	0= IF,	IFEQ,
NE IF,	0= NOT IF,	IFNE,
CS IF,	CS IF,	IFCS,
CC IF,	CS NOT IF,	IFCC,
VS IF,	-------	IFVS,
VC IF,	-------	IFVC,
MI IF,	0< IF,	IFMI,
PL IF,	0< NOT IF,	IFPL,
EQ WHILE,	-------	-----
NE WHILE,	-------	-----
CC WHILE,	-------	-----
CC WHILE,	-------	-----
VS WHILE,	-------	-----
VC WHILE,	-------	-----
MI WHILE,	-------	-----
PL WHILE,	-------	-----
EQ UNTIL,	0= UNTIL,	0= UNTIL,
NE UNTIL,	0= NOT UNTIL,	0= NOT UNTIL,
CS UNTIL,	CS UNTIL,	-----
CC UNTIL,	CS NOT UNTIL,	-----
VS UNTIL,	-------	-----
VC UNTIL,	-------	-----
MI UNTIL,	0< UNTIL,	-----
PL UNTIL,	0< NOT UNTIL,	-----

Chart 2

In all versions, the word END, is synonymous with the word UNTIL,. Likewise, THEN, is synonymous with ENDIF,.

In the valFORTH and Fig assemblers, compiler security is performed to give added protection to the user against assembly errors. To accomplish this, the word C; or its synonym END-CODE is used to terminate the assembly word and perform the check. To remain compatible with APX FORTH, C; is not required in this release of valFORTH. However, it is strongly recommended that C; be used. Although C; and END-CODE are identical, C; is used in-house at Valpar for brevity. (Note that in later releases of valFORTH, C; will become mandatory).

There are several ways in which the indirect jump in the 6502 architecture is implemented in FORTH assemblers. The valFORTH assembler supports three common versions. Thus,

```
                    JMP (VECTOR)
```

can be:

```
                    VECTOR    )JMP,
                    VECTOR  )  JMP,
              or    VECTOR   JMP(),
```

It is recommended that the first version be used.

It must be remembered that valFORTH's additional constructs may not be recognized by assemblers available from other vendors. If assembly listings are to be published for general 6502 FORTH users, it is suggested that valFORTH's advanced features not be used so that novice programmers can still make use of valuable pieces of code.

Appendix A — valFORTH Code Equivalents

This appendix gives the valFORTH assembly code for all 6502 code listings which are marked. Although listing 1 has already been translated to valFORTH assembly code, it is reproduced here for completeness.

Listing 1

```
     DECIMAL
     CODE CLR
          TYA,                    (Move a blank [0] into A)
          # 39 LDY,               (Move count into Y)
          BEGIN,                  (Start looping)
             88 )Y STA,           (Move in a blank)
             DEY,                 (decrement pointer)
          MI UNTIL,               (Go until count < 0)
          NEXT JMP,               (Do a normal re-entry)
     C;
```

Listing 2

```
     CODE 1-                      (Decrement 16 bit value)
          0 ,X LDA,               (Get the low byte)
          NE IF,                  (If a borrow will occur,)
             1 ,X DEC,            (then borrow from high...)
          ENDIF,
          0 ,X DEC,               (Decrement low)
          NEXT JMP,               (Re-enter FORTH)
     C;
```

Listing 3

```
     CODE DUP                     (Duplicate TOS)
          0 ,X LDA,               (Get low byte)
          PHA,                    (Set up for PUSH)
          1 ,X LDA,               (Put high in Accumulator)
          PUSH JMP,               (Push 16 bit value)
     C;
```

Listing 4

```
     CODE SWAP                    (Exchange top stack items)
          2 ,X LDA,               (Get low byte of 2OS)
          PHA,                    (Save it)
          0 ,X LDA,               (Put low byte of TOS)
          2 ,X STA,               (  into low byte of 2OS
          3 ,X LDA,               (Save high byte of 2OS)
          1 ,X LDY,               (Put high byte of TOS)
          3 ,X STY,               (  into high byte of 2OS)
          PUT JMP,                (Put old 2OS into TOS)
     C;
```

Listing 5

```
HEX
CODE ?TERMINAL                  (Any console key pressed?)
     D01F LDA,                  (Load status byte)
     # 7 EOR,                   (Any low bits reset?
     NE IF,                     (If so,)
        INY,                    (then leave a true value)
     ENDIF,
     TYA,                       (Put true or false into A)
     PUSHOA JMP,                (Push to parameter stack)
C; DECIMAL
```

Listing 6

```
CODE C@                         (Byte fetch routine)
     0 X) LDA,                  (Load from address on TOS)
     PUTOA JMP,                 (Push byte value)
C;
```

Listing 7

```
CODE XOR                        (One example of XOR)
     0 ,X LDA,                  (Get low byte of TOS)
     2 ,X EOR,                  (Exclusive or it with 2OS)
     PHA,                       (Push low result)
     1 ,X LDA,                  (Get high byte of TOS)
     3 ,X EOR,                  (XOR it with high of 2OS)
     BINARY JMP,                (Drop TOS and replace 2OS)
C;
```

Listing 8

```
CODE XOR                        (Another exclusive or)
     0 ,X LDA,                  (Get low byte of TOS)
     2 ,X EOR,                  (XOR with low of 2OS)
     2 ,X STA,                  (Put in low of 2OS)
     1 ,X LDA,                  (Get high byte of TOS)
     3 ,X EOR,                  (XOR with high of 2OS)
     3 ,X STA,                  (Put in high of 2OS)
     POP JMP,                   (Drop TOS)
C;
```

Listing 9

```
CODE C!                         (Byte store routine)
     2 ,X LDA,                  (Pick up byte to store)
     0 )X STA,                  (Indirectly store it)
     POPTWO JMP,                (Drop address and byte)
C;
```

Listing 10

```
CODE >R                         (Transfer TOS to R-stack)
     1 ,X LDA,                  (Pick up high of TOS)
     PHA,                       (Put on R-stack)
     0 ,X LDA,                  (Pick up low of TOS)
     PHA,                       (Put on R-stack)
     POP JMP,                   (Lose top stack item)
C;
```

Listing 11

```
HEX
CODE K                          (3rd inner DO loop index)
     XSAVE STX,                 (Save P-stack pointer)
     TSX,                       (Pick up R-stack pointer)
     109 ,X LDA,                (Pick up low byte of value)
     PHA,                       (Save it)
     10A ,X LDA                 (Put high byte of value in A)
     XSAVE LDX,                 (Restore P-stack pointer)
     PUSH JMP,                  (Push 16 bit index value)
C; DECIMAL
```

Appendix B — Quick Reference Chart

valFORTH 6502 Assembly Words

ASSEMBLER (---)

Calls up the assembler vocabulary for subsequent assembly language programming.

CODE cccc (---)

Enters the new word "cccc" into the dictionary as machine language word and calls up the assembler vocabulary for subsequent assembly language programming. CODE also sets the system up for security checking.

C; (---)

Terminates an assembly language definition by performing a security check and setting the CONTEXT vocabulary to the same as the CURRENT vocabulary.

END-CODE (---)

A commonly used synonym for the word C; above. The word C; is recommended over END-CODE.

SUBROUTINE cccc (---)

Enters the new word "cccc" into the dictionary as machine language subroutine and calls up the assembler vocabulary for subsequent assembly language programming. SUBROUTINE also sets the system up for security checking.

;CODE (---)

When the assembler is loaded, puts the system into the assembler vocabulary for subsequent assembly language programming. See main glossary for further explanation.

Control Structures

IF, (flag --- addr 2)

Begins a machine language control structure based on the 6502 status flag on top of the stack. Leaves an address and a security check value for the ELSE, or ENDIF, clauses below. "flag" can be EQ , NE , CC , CS , VC , VS , MI , or PL . Command forms:

```
...flag..IF,..if-true..ENDIF,...all...
...flag..IF,..if-true..ELSE,..if-false..ENDIF,..all...
```

ELSE, (addr 2 --- addr 3)

Used in an IF, clause to allow for execution of code only if IF, clause is false. If the IF, clause is true, this code is bypassed. See IF, above for command form.

ENDIF, (addr 2/3 ---)

Used to terminate an IF, control structure clause. Additionally, ENDIF, resolves all forward references. See IF, above for command form.

BEGIN, (--- addr 1)

Begins machine language control structures of the following forms:

```
...BEGIN,...AGAIN,...
...BEGIN,...flag..UNTIL,...
...BEGIN,...flag..WHILE,..while-true..REPEAT,...
```

where "flag" is one of the 6502 statuses: EQ , NE , CC , CS , VC , VS , MI , and PL . See the very similar BEGIN in the main glossary for additional information.

UNTIL, (addr 1 flag ---)

Used to terminate a post-testing BEGIN, clause thus allowing for conditional looping of a program segment while "flag" is false. See BEGIN, above for more information.

WHILE, (addr 1 flag --- addr 4)

Used to begin a pre-testing BEGIN, clause thus allowing for conditional looping of a program segment while"flag" is true. See BEGIN, above for command format.

REPEAT, (addr 4 ---)

Used to terminate a pre-testing BEGIN,..WHILE, clause. Additionally, REPEAT, resolves all forward addresses of the current WHILE, clause. See BEGIN, above.

AGAIN, (addr 1 ---)

Used to terminate an unconditional BEGIN, clause. Execution cannot exit this loop unless a JMP, instruction is used. See BEGIN, clause for more information.

Parameter Passing

NEXT (--- addr)

Transfers control to the next FORTH word to be executed. The parameter stack is left unchanged.

PUSH (--- addr)

Pushes a 16 bit value to the parameter stack whose low byte is found on the 6502 return stack and whose high byte is found in the accumulator.

PUSHOA (--- addr)

Pushes a 16 bit value to the parameter stack whose low byte is found in the accumulator and whose high byte is zero.

PUT (--- addr)

Replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is found on the 6502 stack and whose high byte is in the accumulator.

PUTOA (--- addr)

Replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is in the accumulator and whose high byte is set to zero.

BINARY (--- addr)

Drops the top value of the parameter stack and then performs a PUT operation described above.

POP and POPTWO (--- addr)

POP drops one value from the parameter stack. POPTWO drops two values from the parameter stack.

SETUP (--- addr)

Moves one to four values to the N scratch area in the zero page and drops all values moved from the parameter stack.

N (--- addr)

Points to a nine-byte scratch area in the zero page beginning at N-1 and going to N+7. Typically used by words which use indirect-indexed addressing where addresses must be stored in the zero page. See SETUP above.

Opcodes (---)

```
ADC,  AND,  ASL,  BIT,  BRK,  CLC,  CLD,  CLI,
CLV,  CMP,  CPX,  CPY,  DEC,  DEX,  DEY,  EOR,
INC,  INX,  INY,  JSR,  JMP,  LDA,  LDX,  LDY,
LSR,  NOP,  ORA,  PHA,  PHP,  PLA,  PLP,  ROL,
ROR,  RTI,  RTS,  SBC,  SEC,  SED,  SEI,  STA,
STX,  TAX,  TAY,  TSX,  TXA,  TXS,  TYA,
```

Aliases

```
NXT,   =  NEXT JMP,
PSH,   =  PUSH JMP,
PUT,   =  PUT JMP,
PSHA,  =  PUSHOA JMP,
PUTA,  =  PUTOA JMP,
POP,   =  POP JMP,
POP2,  =  POPTWO JMP,
XL,    =  XSAVE LDX,
XS,    =  XSAVE STX,
THEN,  =  ENDIF,
END,   =  UNTIL,
```

```
Screen:  30
 0 ( Auto command                    )
 1
 2 BASE @ HEX
 3
 4 : ZAP                  ( addr # -- )
 5   -DUP
 6   IF 0+S
 7    DO D20A C@ 7F AND I C!
 8    LOOP
 9   ELSE DROP
10   ENDIF ;
11
12 : -WAIT                       ( -- )
13    BEGIN ?TERMINAL NOT UNTIL ;
14
15 DCX                              ==>
```

```
Screen:  31
 0 ( Auto command                    )
 1
 2 : BEHEAD                       ( -- )
 3   0 )R CR ." Now protecting..."
 4   CR VOC-LINK @
 5   BEGIN
 6    DUP 2- @
 7    BEGIN
 8     DUP 1+ OVER R) 1+ )R
 9     R 15 MOD NOT IF ." ." ENDIF
10     R 495 MOD NOT IF CR ENDIF
11     C@ 63 AND WIDTH @ MIN 1-
12     ZAP PFA LFA @ DUP NOT
13    UNTIL
14    DROP @ DUP NOT
15   UNTIL R) 2DROP ;                --)
```

```
Screen:  32
 0 ( Auto command                    )
 1
 2 : QUEST                        ( -- )
 3   CR
 4   ." Save on disk:          "
 5   ." press OPTION" CR
 6   ." Save on cassette:      "
 7   ." press SELECT" CR
 8   ." Exit:                  "
 9   ." press START" CR CR ;
10
11 : DODISK                      ( -- )
12   (SAVE) ' SAVE 32 + @EX
13   741 @ 128 - 1 1 R/W ;
14
15                                  ==>
```

```
Screen:  33
 0 ( Auto command                    )
 1
 2 : QUEST2                   ( -- n )
 3   ." Format and save:       "
 4   ." press OPTION" CR
 5   ." Just save:             "
 6   ." press SELECT" CR CR
 7   WAIT ?TERMINAL -WAIT
 8   ." Prepare disk -- "
 9   ." press START"
10   WAIT -WAIT ;
11 : CSV                         ( -- )
12   ." Prepare cassette "
13   ." (play/record) -- " CR
14   ." press START"  CR
15   WAIT CSAVE -WAIT ;              --)
```

```
Screen:  34
 0 ( Auto command                    )
 1
 2 : DSV
 3   QUEST2
 4   4 =
 5   IF
 6    1 (FMT) 1 <>
 7    IF
 8     CR ." Format error"
 9    ELSE
10     DODISK
11    ENDIF
12   ELSE
13    DODISK
14   ENDIF CR ;
15                                  ==>
```

```
Screen:  35
 0 ( Auto command                    )
 1
 2 : DECIS                        ( -- )
 3   BEGIN
 4    QUEST WAIT ?TERMINAL -WAIT
 5    DUP 1 =
 6    IF DROP 1
 7    ELSE
 8     2 =
 9     IF
10      CSV
11     ELSE
12      DSV
13     ENDIF 0
14    ENDIF
15   UNTIL ;                         --)
```

```
Screen:  36
 0 ( Auto command                    )
 1
 2 : AUTO                         ( -- )
 3   [COMPILE] ' CR
 4   ." Auto? Y/N " KEY 89 = CR
 5   IF
 6    CFA ' ABORT 6 + !
 7    ' COLD CFA ' ABORT 8 + !
 8    -1 26 +ORIGIN !
 9    ' ZAP NFA DP !
10    BEHEAD DECIS ABORT
11   ELSE
12    DROP ." Auto aborted..." CR
13   ENDIF ;                 BASE !
14
15
```

```
Screen:  37
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  38
 0 ( Text output:  S:  P:            )
 1
 2 BASE @ HEX
 3
 4 : S:                           ( f -- )
 5   PFLAG @ SWAP
 6   IF 1 OR ELSE FE AND ENDIF
 7   PFLAG ! ;
 8
 9 : P:                           ( f -- )
10   PFLAG @ SWAP
11   IF 2 OR ELSE FD AND ENDIF
12   PFLAG ! ;
13
14
15                                    ==>
```

```
Screen:  39
 0 ( Text output:  BEEP  ASCII       )
 1
 2 : BEEP                         ( -- )
 3   0C0 0
 4   DO
 5     08 0D01F C! 6 0 DO LOOP
 6     00 0D01F C! 6 0 DO LOOP
 7   LOOP ;
 8
 9 : ASCII                 ( ccc, -- <b>)
10   BL WORD
11   HERE 1+ C@
12   STATE @
13   IF
14     COMPILE CLIT C,
15   ENDIF ;  IMMEDIATE              -->
```

```
Screen:  40
 0 ( Text output:  EJECT  LISTS      )
 1 DCX
 2
 3 : EJECT                        ( -- )
 4   12 EMIT ;
 5
 6 : LISTS                    ( s # -- )
 7   0 <ROT 0+S
 8   DO
 9     CR I LIST
10     1+ DUP 3 MOD 0=
11     IF EJECT ENDIF
12     ?EXIT
13   LOOP
14   DROP ;
15                                    ==>
```

```
Screen:  41
 0 ( Text output:  PLISTS  PLIST     )
 1
 2 : PLISTS                   ( s # -- )
 3   PFLAG @ <ROT
 4   ON P:
 5   LISTS
 6   CR PFLAG ! ;
 7
 8 : PLIST                      ( s -- )
 9   1 PLISTS ;
10
11
12
13
14
15                          BASE !
```

```
Screen:  42
 0 ( Debug:  B?  [FREE]  FREE      )
 1 BASE @ DCX
 2 '( S: )( 19 KLOAD )
 3 HEX
 4
 5 : B?                          ( -- )
 6   BASE @ DUP            ( Display )
 7   DECIMAL .             ( current )
 8   BASE ! ;              ( radix )
 9
10 : (FREE)                    ( -- n )
11   2E5 @ PAD - ;
12
13 : FREE                          ( -- )
14   (FREE) U. ." bytes" CR ;
15                                  ==>
```

```
Screen:  43
 0 ( Debug:  H.  CFALIT            )
 1
 2 : CFALIT            ( ccc, -- <b>)
 3   STATE @
 4   [COMPILE] [
 5   [COMPILE] ' CFA
 6   SWAP IF [COMPILE] ] ENDIF
 7   [COMPILE] LITERAL ;
 8   IMMEDIATE
 9
10 '( H. --> )(  )
11 : H.                        ( b -- )
12   BASE @ HEX            ( Display )
13   SWAP 0                ( # in hex )
14   <# # # #> TYPE
15   BASE ! ;                      -->
```

```
Screen:  44
 0 ( Debug:  #DUMP                 )
 1
 2 ( memory dump  )
 3
 4 : #DUMP                  ( a # -- )
 5   0+S
 6   DO
 7     CR I 5 U.R I
 8     DUP 8 + SWAP
 9     DO
10       I C@ 4 .R
11     LOOP
12     ?EXIT
13   8 /LOOP
14   CR ;
15                                  ==>
```

```
Screen:  45
 0 ( Debug:  CDUMP                 )
 1
 2 ( Character dump routine        )
 3
 4 : CDUMP                  ( a # -- )
 5   PFLAG @ <ROT OFF P:
 6   OVER + SWAP
 7   DO
 8     CR I 5 U.R
 9     SPACE I DUP 10 + SWAP
10     DO
11       I C@ SPEMIT
12     LOOP
13     ?EXIT
14   10 /LOOP
15   CR PFLAG ! ;                   -->
```

```
Screen:  46
 0 ( Stack print:  DEPTH           )
 1
 2 : DEPTH                  ( n -- )
 3   S0 @ SP@ - 2/ 1- ;
 4
 5 CFALIT . VARIABLE X.S
 6
 7 : XDOTS                       ( -- )
 8   DEPTH
 9   IF
10     SP@ 2- S0 @ 2-
11     DO I @ X.S @ EXECUTE
12     -2 +LOOP
13   ELSE
14     ." Stack empty "
15   ENDIF ;                        ==>
```

```
Screen:  47
 0 ( Stack print:  .S  U.S  STACK )
 1
 2 : .S    CFALIT  . X.S ! XDOTS ;
 3 : U.S   CFALIT U. X.S ! XDOTS ;
 4
 5 : STKPRT
 6   CR ." ( " XDOTS ." ) " ;
 7
 8 : STACK                  ( f -- )
 9   IF
10    CFALIT STKPRT
11   ELSE
12    CFALIT NOOP
13   ENDIF
14   [ ' PROMPT 11 + ]
15   LITERAL ! ;                    -->
```

```
Screen:  48
  0 ( FORTH colon decompiler      )
  1
  2
  3 0 VARIABLE .WORD
  4
  5 : PWORD
  6   2+ NFA ID. ;
  7
  8 : 1BYTE
  9   PWORD .WORD @ C@ .
 10   1 .WORD +! ;
 11
 12 : 1WORD
 13   PWORD .WORD @ @ .
 14   2 .WORD +! ;
 15                              ==>
```

```
Screen:  49
  0 ( FORTH colon decompiler      )
  1 : NP                  ( n -- n )
  2   DUP CFALIT ;S = OVER
  3   CFALIT (;CODE) = OR
  4   IF PWORD CR CR PROMPT QUIT
  5   ENDIF ?TERMINAL
  6   IF DROP PROMPT QUIT ENDIF ;
  7
  8 : BRNCH
  9   PWORD ." to " .WORD @ .WORD @
 10   @ + U. 2 .WORD +! ;
 11
 12 : NXT1
 13   .WORD @ U. 2 SPACES
 14   .WORD @ @ 2 .WORD +! ;
 15                              -->
```

```
Screen:  50
  0 ( FORTH colon decompiler      )
  1
  2 : STG
  3   PWORD 22 EMIT .WORD @
  4   DUP COUNT TYPE 22 EMIT
  5   C@ .WORD @ + 1+ .WORD ! ;
  6
  7 : CKIT
  8   DUP  CFALIT 0BRANCH =
  9   OVER CFALIT BRANCH  = OR
 10   OVER CFALIT (LOOP)  = OR
 11   OVER CFALIT (+LOOP) = OR
 12   OVER CFALIT (/LOOP) = OR
 12
 13   IF BRNCH
 14   ELSE DUP CFALIT LIT =
 15     IF 1WORD                 ==>
```

```
Screen:  51
  0 ( FORTH colon decompiler      )
  1
  2     ELSE
  3       DUP CFALIT CLIT =
  4       IF 1BYTE
  5       ELSE
  6         DUP CFALIT COMPILE =
  7         IF PWORD CR NXT1 PWORD
  8         ELSE
  9           DUP 4 - @ A922 =
 10           IF STG
 11           ELSE PWORD ENDIF
 12         ENDIF
 13       ENDIF
 14     ENDIF
 15   ENDIF ;                    -->
```

```
Screen:  52
  0 ( FORTH colon decompiler      )
  1
  2 : ?DOCOL
  3   DUP 2- @
  4   [ ' : 12 + ] LITERAL -
  5   IF ." Primitive pfa dump:"
  6     2- @ 18 #DUMP
  7     PROMPT QUIT
  8   ENDIF ;
  9
 10
 11
 12
 13
 14
 15                              ==>
```

```
Screen:  53
  0 ( FORTH colon decompiler      )
  1
  2 : DCMPR                ( PFA -- )
  3   DUP NFA CR CR DUP ID.
  4   C@ 40 AND
  5   IF ." (IMMEDIATE)"
  6   ENDIF
  7   CR CR ?DOCOL .WORD !
  8   BEGIN NXT1 NP CKIT CR AGAIN ;
  9
 10 : DECOMP
 11   [COMPILE] ' DCMPR ;
 12
 13
 14
 15                          BASE ! ;S
```

```
Screen:  54
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  55
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  56
 0 ( fig editor:  TEXT  LINE      )
 1
 2 BASE @  HEX
 3
 4 ( This editor is based on the  )
 5 ( example editor supplied in   )
 6 ( the "fig-FORTH Installation  )
 7 ( Manual."                     )
 8
 9 : TEXT
10   HERE C/L 1+ BLANKS WORD
11   HERE PAD C/L 1+ CMOVE ;
12
13 : LINE
14   DUP FFF0 AND 17 ?ERROR
15   SCR @ (LINE) DROP ;         ==>
```

```
Screen:  57
 0 ( fig editor:  MARK            )
 1
 2 : MARK
 3   10 0
 4   DO
 5     I LINE UPDATE
 6     DROP
 7   LOOP ;
 8
 9
10
11
12
13
14
15                                -->
```

```
Screen:  58
 0 ( fig editor:  WHERE           )
 1
 2 VOCABULARY EDITOR IMMEDIATE
 3
 4 ( Note: the fig bug is fixed )
 5 (       in WHERE below.      )
 6
 7 : WHERE                ( n n -- )
 8   2DUP DUP B/SCR / DUP SCR !
 9   ." Scr # " DECIMAL . SWAP
10   C/L /MOD C/L * ROT BLOCK +
11   CR C/L -TRAILING TYPE
12   CR HERE C@ - 2- 0 MAX SPACES
13   1 2FE C! 1C EMIT 0 2FE C!
14   [COMPILE] EDITOR QUIT ;
15                               ==>
```

```
Screen:  59
 0 ( fig editor:  #L's  -MOVE     )
 1
 2 EDITOR DEFINITIONS
 3
 4 : #LOCATE              ( -- n n )
 5   R# @ C/L /MOD ;
 6
 7 : #LEAD                ( -- n n )
 8   #LOCATE LINE SWAP ;
 9
10 : #LAG                 ( -- n n )
11   #LEAD DUP )R + C/L R) - ;
12
13 : -MOVE                ( n n -- )
14   LINE C/L CMOVE UPDATE ;
15                                -->
```

```
Screen:  60
  0 ( fig editor:  H  E  S                )
  1
  2 : H                           ( n -- )
  3   LINE PAD 1+ C/L
  4   DUP PAD C! CMOVE ;
  5
  6 : E                           ( n -- )
  7   LINE C/L BLANKS UPDATE ;
  8
  9 : S                           ( n -- )
 10   DUP 1- 0E
 11   DO
 12     I LINE
 13     I 1+ -MOVE
 14   -1 +LOOP
 15   E ;                              ==>
```

```
Screen:  61
  0 ( fig editor:  D  M                  )
  1
  2 : D                           ( n -- )
  3   DUP H 0F DUP ROT
  4   DO
  5     I 1+ LINE
  6     I -MOVE
  7   LOOP
  8   E ;
  9
 10 : M                           ( n -- )
 11   R# +! CR SPACE
 12   #LEAD TYPE 14 EMIT #LAG
 13   TYPE #LOCATE . DROP ;
 14
 15                                    -->
```

```
Screen:  62
  0 ( fig editor:  T  L  R  P            )
  1
  2 : T                           ( n -- )
  3   DUP C/L * R# !
  4   DUP H 0 M ;
  5
  6 : L                             ( -- )
  7   SCR @ LIST 0 M ;
  8
  9 : R                           ( n -- )
 10   PAD 1+ SWAP -MOVE ;
 11
 12 : P                           ( n -- )
 13   1 TEXT R ;
 14
 15                                    ==>
```

```
Screen:  63
  0 ( fig editor:  I  TOP                )
  1
  2 : I                           ( n -- )
  3   DUP S R ;
  4
  5 : TOP                           ( -- )
  6   0 R# ! ;
  7
  8
  9
 10
 11
 12
 13
 14
 15                                    -->
```

```
Screen:  64
  0 ( fig editor:  CLEAR  COPY           )
  1
  2 : CLEAR                       ( n -- )
  3   SCR ! 10 0
  4   DO
  5     FORTH I EDITOR E
  6   LOOP ;
  7
  8 : COPY                      ( n n -- )
  9   B/SCR * OFFSET @ + SWAP
 10   B/SCR * B/SCR OVER + SWAP
 11   DO
 12     DUP FORTH I BLOCK
 13     2- ! 1+ UPDATE
 14   LOOP
 15   DROP ;                           ==>
```

```
Screen:  65
  0 ( fig editor:  1LINE  FIND           )
  1
  2 : 1LINE                       ( -- f )
  3   #LAG PAD COUNT
  4   MATCH R# +! ;
  5
  6 : FIND                          ( -- )
  7   BEGIN
  8     3FF R# @ <
  9     IF
 10       TOP PAD HERE C/L
 11       1+ CMOVE 0 ERROR
 12     ENDIF
 13     1LINE
 14   UNTIL ;
 15                                    -->
```

```
Screen:  66
 0 ( fig editor:  DELETE                )
 1
 2 : DELETE                    ( n -- )
 3   )R #LAG + FORTH R -
 4   #LAG R MINUS R# +! #LEAD
 5   + SWAP CMOVE R) BLANKS
 6   UPDATE ;
 7
 8
 9
10
11
12
13
14
15                                 ==)
```

```
Screen:  67
 0 ( fig editor:  N  F  B  X            )
 1
 2 : N                         ( -- )
 3   FIND 0 M ;
 4
 5 : F                         ( -- )
 6   1 TEXT N ;
 7
 8 : B                         ( -- )
 9   PAD C@ MINUS M ;
10
11 : X                         ( -- )
12   1 TEXT FIND
13   PAD C@ DELETE
14   0 M ;
15                                 --)
```

```
Screen:  68
 0 ( fig editor:  TILL  C               )
 1
 2 : TILL                      ( -- )
 3   #LEAD + 1 TEXT
 4   1LINE 0= 0 ?ERROR
 5   #LEAD + SWAP -
 6   DELETE 0 M ;
 7
 8 : C
 9   1 TEXT PAD COUNT #LAG ROT
10   OVER MIN )R FORTH R R# +!
11   R - )R DUP HERE R CMOVE
12   HERE #LEAD + R) CMOVE R)
13   CMOVE UPDATE 0 M ;
14
15                                 ==)
```

```
Screen:  69
 0 ( End of fig-FORTH editor            )
 1
 2 FORTH DEFINITIONS DCX
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15                          BASE !
```

```
Screen:  70
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  71
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  72
 0 ( Disk copy routines                  )
 1 BASE @ DCX
 2
 3 0 VARIABLE SEC/PAS
 4 0 VARIABLE SECNT
 5
 6 : AXLN                    ( system )
 7   4 PICK 0
 8   DO 3 PICK I 128 * +
 9       3 PICK I + 3 PICK R/W
10   LOOP 2DROP 2DROP ;
11
12 : DCSTP
13   741 @ PAD DUP 1 AND - -
14   0 128 U/ SWAP DROP
15   SEC/PAS ! 0 SECNT ! ;          ==>
```

```
Screen:  73
 0 ( Disk copy routines                  )
 1
 2 : DISKCOPY1                  ( -- )
 3   DCSTP
 4   BEGIN
 5    CR CR ." Insert source and pu
 6 sh START" WAIT
 7    720 SECNT @ - SEC/PAS @ MIN
 8    DUP )R PAD DUP 1 AND - SECNT
 9    @ 2DUP 5 PICK (ROT 1 AXLN
10    CR CR ." Insert dest.  and pu
11 sh START" WAIT 0 AXLN CR
12    R) SECNT +! SECNT @ DUP .
13    ." sectors copied" 720 =
14   UNTIL EMPTY-BUFFERS
15   CR ." Done" CR ;               -->
```

```
Screen:  74
 0 ( Disk copy routines                  )
 1
 2 : DISKCOPY2                  ( -- )
 3   DCSTP
 4   CR ." Insert source in drive 1
 5 " CR ." Insert dest.  in drive 2
 6 " CR ." Press START to copy"
 7   WAIT
 8   BEGIN
 9    720 SECNT @ - SEC/PAS @ MIN
10    DUP )R PAD DUP 1 AND - SECNT
11    @ 2DUP 5 PICK (ROT
12    1 AXLN 720 + 0 AXLN
13    R) SECNT +! SECNT @ 720 =
14   UNTIL EMPTY-BUFFERS
15   CR ." Done" CR ;    BASE !
```

```
Screen:  75
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  76
 0 ( 6502 Assembler in FORTH             )
 1 (
 2 ( Originally written by
 3 ( Patrick Mullarky.
 4 (
 5 ( Modified extensively 2/82
 6 ( by Stephen Maguire,
 7 ( Valpar International
 8 (
 9 (
10 ( This assembler conforms to the
11 ( fig "INSTALLATION GUIDE" and
12 ( to APX versions of FORTH.
13 (
14 (  )
15                                   ==>
```

```
Screen:  77
 0 ( 6502 Assembler in FORTH             )
 1 (
 2 ( Now supports:
 3 (
 4 ( IF,...ELSE,...ENDIF,
 5 ( BEGIN,...WHILE,...REPEAT,
 6 ( BEGIN,...AGAIN,
 7 ( BEGIN,...any flag UNTIL,
 8 ( C; & END-CODE
 9 ( ;CODE
10 (
11 ( Also supports:
12 (
13 ( compiler security
14 ( definition checking                )
15                                   -->
```

```
Screen:  78
  0 ( 6502 Assembler in FORTH       )
  1 '( TRANSIENT TRANSIENT )(   )
  2 BASE @ HEX
  3 ASSEMBLER DEFINITIONS
  4
  5 : SB
  6   (BUILDS C, DOES) @ C, ;
  7
  8  000 SB BRK,     018 SB CLC,
  9  0D8 SB CLD,     058 SB CLI,
 10  0B8 SB CLV,     0CA SB DEX,
 11  088 SB DEY,     0E8 SB INX,
 12  0C8 SB INY,     0EA SB NOP,
 13  048 SB PHA,     008 SB PHP,
 14  068 SB PLA,     028 SB PLP,
 15  040 SB RTI,     060 SB RTS,   ==)
```

```
Screen:  79
  0 ( 6502 Assembler in FORTH       )
  1  038 SB SEC,     0F8 SB SED,
  2  078 SB SEI,     0AA SB TAX,
  3  0BA SB TSX,     08A SB TXA,
  4  09A SB TXS,     098 SB TYA,
  5  0A8 SB TAY,
  6
  7 0 VARIABLE )J  : ) 1 )J ! ;
  8
  9 : 3BY
 10   (BUILDS C, DOES) C@ DUP 4C =
 11   IF )J @ IF DROP 6C ENDIF
 12   ENDIF C, , 0 )J ! ;
 13
 14  04C 3BY JMP,   06C 3BY JMP(),
 15  020 3BY JSR,   06C 3BY )JMP,  --)
```

```
Screen:  80
  0 ( 6502 Assembler in FORTH       )
  1
  2 : 256(  DUP 100 ( HEX ) U( ;
  3
  4 70 CONSTANT VC   ( over clear )
  5 50 CONSTANT VS   ( over set )
  6 B0 CONSTANT CC   ( carry clear )
  7 90 CONSTANT CS   ( carry set )
  8 D0 CONSTANT EQ   ( zero )
  9 F0 CONSTANT NE   ( non-zero )
 10 30 CONSTANT PL   ( positive )
 11   1 ?PAIRS 4C C, , ;  IMMEDIATE
 12
 13 : IF,
 14   C, 0 C, HERE 2 ;     IMMEDIATE
 15                                ==)
```

```
Screen:  81
  0 ( 6502 Assembler in FORTH       )
  1
  2 : ENDIF,
  3   DUP 2 = IF
  4     DROP DUP HERE SWAP -
  5     DUP  7F ) 5 ?ERROR
  6     DUP -80 ( 5 ?ERROR
  7     SWAP 1- C!
  8   ELSE
  9     3 ?PAIRS HERE SWAP !
 10   ENDIF ;              IMMEDIATE
 11
 12 : ELSE,
 13   DUP 2 ?PAIRS 4C C, HERE 0 ,
 14   (ROT [COMPILE] ENDIF, 3 ;
 15                                --)
```

```
Screen:  82
  0 ( 6502 Assembler in FORTH       )
  1
  2 : THEN,
  3   [COMPILE] ENDIF, ;  IMMEDIATE
  4
  5 : BEGIN,
  6   HERE 1 ;             IMMEDIATE
  7
  8 : UNTIL,
  9   SWAP 1 ?PAIRS C,
 10   HERE 1+ - DUP -80
 11   ( 5 ?ERROR C, ;      IMMEDIATE
 12
 13 : END,
 14   [COMPILE] UNTIL, ;  IMMEDIATE
 15                                ==)
```

```
Screen:  83
  0 ( 6502 Assembler in FORTH       )
  1
  2 : WHILE,
  3   SWAP 1 ?PAIRS [COMPILE] IF,
  4   DROP 4 ;             IMMEDIATE
  5
  6 : REPEAT,
  7   4 ?PAIRS SWAP 4C C, , 2
  8   [COMPILE] ENDIF, ;  IMMEDIATE
  9
11 10 CONSTANT MI   ( negative )

 12
 13 : END-CODE
 14   [COMPILE] C; ;       IMMEDIATE
 15                                --)
```

```
Screen:  84
  0 ( 6502 Assembler in FORTH       )
  1 0D VARIABLE MODE    ( ABS mode )
  2 00 VARIABLE ACC     ( A-reg? )
  3
  4 : BIT,
  5   256< IF 24 C, C,
  6   ELSE 2C C, , ENDIF ;
  7
  8 : CKMODE
  9   MODE @ =
 10   IF           ( MODE = MODE - 8 )
 11     256<       ( if addr < 256   )
 12     IF
 13       -08 MODE +!
 14     ENDIF
 15   ENDIF ;                    ==>
```

```
Screen:  85
  0 ( 6502 Assembler in FORTH       )
  1
  2 : M0
  3   <BUILDS
  4     C,
  5   DOES>
  6     SWAP 0D CKMODE
  7     1D CKMODE SWAP
  8     C@ MODE @ OR C,
  9     256< IF C, ELSE , ENDIF
 10     0D MODE ! ;    ( ABS mode )
 11
 12   00 M0 ORA,      20 M0 AND,
 13   40 M0 EOR,      60 M0 ADC,
 14   80 M0 STA,      A0 M0 LDA,
 15   C0 M0 CMP,      E0 M0 SBC,   -->
```

```
Screen:  86
  0 ( 6502 Assembler in FORTH       )
  1 : !ADDR   C, 256< IF C, ELSE ,
  2   ENDIF 0D MODE ! ;
  3
  4 : ZPAGE
  5   OVER 100 U< IF F7 AND ENDIF ;
  6
  7 : M1
  8   <BUILDS C, DOES> C@ ACC @
  9   IF FB AND C, ELSE MODE @ 1D =
 10   IF 10 ELSE 0 ENDIF OR ZPAGE
 11   !ADDR ENDIF 0 ACC ! ;
 12
 13  00E M1 ASL,     02E M1 ROL,
 14  04E M1 LSR,     06E M1 ROR,
 15  0CE M1 DEC,     0EE M1 INC,   ==>
```

```
Screen:  87
  0 ( 6502 Assembler in FORTH       )
  1
  2 : OPCODE
  3   C@ ZPAGE MODE @ 1D =
  4   MODE @ 19 = OR
  5   IF 10 OR ENDIF ;
  6
  7 : M2
  8   <BUILDS C,
  9   DOES> OPCODE MODE @ 9 =
 10   IF 4 - ENDIF !ADDR ;
 11
 12 : M3
 13   <BUILDS C,
 14   DOES> OPCODE !ADDR ;
 15                             -->
```

```
Screen:  88
  0 ( 6502 Assembler in FORTH       )
  1   0AC M2 LDY,     0AE M2 LDX,
  2   0CC M2 CPY,     0EC M2 CPX,
  3   08C M3 STY,     08E M3 STX,
  4
  5 : X)   01 MODE ! ;   ( [addr,X] )
  6 : #    09 MODE ! ;   ( immediate )
  7 : )Y   11 MODE ! ;   ( [addr],Y  )
  8 : ,X   1D MODE ! ;   ( addr,X )
  9 : ,Y   19 MODE ! ;   ( addr,Y )
 10 : .A   01 ACC  ! ;   ( a - reg )
 11
 12 0A SB ASL.A,    2A SB ROL.A,
 13 4A SB LSR.A,    6A SB ROR.A,
 14
 15                             ==>
```

```
Screen:  89
  0 ( 6502 Assembler in FORTH       )
  1
  2 : IFVC, VC [COMPILE] IF, ;
  3 : IFVS, VS [COMPILE] IF, ;
  4 : IFCC, CC [COMPILE] IF, ;
  5 : IFCS, CS [COMPILE] IF, ;
  6 : IFEQ, EQ [COMPILE] IF, ;
  7 : IFNE, NE [COMPILE] IF, ;
  8 : IFPL, PL [COMPILE] IF, ;
  9 : IFMI, MI [COMPILE] IF, ;
 10
 11 : 0= EQ ; : 0<  MI ; : >= EQ ;
 12 : NOT 20 XOR ;   : RP) 101 ,X ;
 13 : BOT 0 ,X ; : SEC 2 ,X ;
 14
 15                             -->
```

```
Screen:  90
 0 ( End of 6502 assembler          )
 1 HEX
 2 : XS,     XSAVE STX, ;
 3 : XL,     XSAVE LDX, ;
 4 : NXT,    NEXT JMP, ;
 5 : POP,    POP JMP, ;
 6 : POP2,   POPTWO JMP, ;
 7 : PSH,    PUSH JMP, ;
 8 : PSHA,   PUSH0A JMP, ;
 9 : PUT,    PUT JMP, ;
10 : PUTA,   PUT0A JMP, ;
11
12
13 FORTH DEFINITIONS
14 '( PERMANENT PERMANENT )(   )
15                          BASE !
```

```
Screen:  91
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  92
 0 ( FORMAT                         )
 1 BASE @ HEX
 2
 3 : FORMAT
 4   CR CR ." Enter Drive#: " KEY
 5   DUP EMIT 30 - 1 MAX 4 MIN CR
 6  ." Hit RETURN to format drive "
 7   DUP . CR
 8  ." Hit any other key to abort "
 9   KEY 9B =
10   IF (FMT) 1 = CR CR ." Format "
11    IF ." OK" ELSE ." ERROR"
12    ENDIF
13   ELSE CR ." Format aborted..."
14     DROP
15   ENDIF CR CR ;        BASE !
```

```
Screen:  93
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  94
 0 ( Buffer relocation              )
 1 BASE @ DCX
 2
 3 : RELOCBUFS           ( addr -- )
 4   DUP 1 AND
 5   IF CR ." Odd buffer address."
 6      CR ." Try again." DROP QUIT
 7   ENDIF
 8   DUP         ' FIRST !
 9   DUP 2112 + ' LIMIT !
10   DUP           PREV  !
11   DUP           USE   !
12   MTB CR 156 EMIT 156 EMIT
13   ." Buffers relocated to "
14   U.  ." and emptied" CR ;
15                                  ==>
```

```
Screen:  95
 0 ( Buffer relocation              )
 1
 2 CR CR ." The buffers take 2112 b
 3 ytes decimal." CR
 4 CR ." To relocate buffers, put t
 5 he new addr on stack and do:" CR
 6  CR 7 SPACES ." RELOCBUFS FORGET
 7  RELOCBUFS" CR CR   BASE !
 8
 9
10
11
12
13
14
15
```

```
Screen:  96
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  97
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  98
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen:  99
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 100
 0 ( Colors:  hue CONSTANTs          )
 1
 2 BASE @ DCX
 3
 4   0 CONSTANT GREY
 5   1 CONSTANT GOLD
 6   2 CONSTANT ORNG
 7   3 CONSTANT RDORNG
 8   4 CONSTANT PINK
 9   5 CONSTANT LVNDR
10   6 CONSTANT BLPRPL
11   7 CONSTANT PRPLBL
12   8 CONSTANT BLUE
13   9 CONSTANT LTBLUE
14  10 CONSTANT TURQ
15  11 CONSTANT GRNBL                ==>
```

```
Screen: 101
 0 ( Colors:  hue CONSTANTs          )
 1
 2  12 CONSTANT GREEN
 3  13 CONSTANT YLWGRN
 4  14 CONSTANT ORNGRN
 5  15 CONSTANT LTORNG
 6
 7
 8
 9
10
11
12
13
14
15                          BASE !  --)
```

```
Screen: 102
 0 ( Colors:  SETCOLOR  BOOTCOLOR )
 1 BASE @ DCX
 2
 3 : SETCOLOR       ( # hue lum -- )
 4   SWAP 16 * OR SWAP
 5   708 ( COLPF0 ) + C! ;
 6
 7 : SE.  SETCOLOR ;
 8
 9 : BOOTCOLOR        ( hue lum -- )
10   SWAP 16 * DUP 4 + DUP
11   [ ' COLD 35 + ] LITERAL C!
12   710 C! OR DUP
13   [ ' COLD 40 + ] LITERAL C!
14   709 C! ;
15                          BASE !
```

```
Screen: 103
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 104
 0 ( Graphics:  CGET                 )
 1
 2 BASE @ DCX '( )SCD )( 68 KLOAD )
 3
 4 HEX
 5 CODE CGET                  ( -- b )
 6   B5 C, 00 C, 48 C,
 7   86 C, XSAVE C,
 8   A2 C, 30 C, A9 C, 07 C,
 9   9D C, 342 , 98 C,
10   9D C, 348 , 9D C, 349 ,
11   68 C, 20 C, CIO ,
12   A6 C, XSAVE C,
13   4C C, PUSH0A ,
14 C;
15                                ==>
```

```
Screen: 105
 0 ( Graphics:  COLOR  POS.  LOC. )
 1
 2 0 VARIABLE CLRBYT
 3
 4 : COLOR                  ( b -- )
 5   CLRBYT ! ;
 6
 7 : POS.                   ( h v -- )
 8   54 C! 55 ! ;
 9
10 : POSITION POS. ;        ( h v -- )
11
12 : LOC.                ( x y -- b )
13   POS. CGET ;
14
15                                -->
```

```
Screen: 106
 0 ( Graphics:  CPUT                 )
 1
 2 HEX
 3 CODE CPUT                  ( b -- )
 4   B5 C, 00 C, 48 C,
 5   86 C, XSAVE C,
 6   A2 C, 30 C, A9 C, 0B C,
 7   9D C, 342 , 98 C,
 8   9D C, 348 , 9D C, 349 ,
 9   68 C, 20 C, CIO ,
10   A6 C, XSAVE C,
11   4C C, POP ,
12 C;
13
14
15                                ==>
```

```
Screen: 107
 0 ( Graphics:  POS@  POSIT  PLOT )
 1
 2 : POS@                   ( -- h v )
 3   55 @ 54 C@ ;
 4
 5 : POSIT                  ( h v -- )
 6   POS. 54 C@ 5A C!
 7        55  @ 5B  ! ;
 8
 9 : PLOT                ( b h v -- )
10   POS. CLRBYT C@ CPUT ;
11
12
13
14
15                                -->
```

```
Screen: 108
  0 ( Graphics:  GTYPE                  )
  1
  2 : GTYPE              ( cnt adr -- )
  3   0 MAX -DUP
  4   IF 0+S
  5    DO I C@ )SCD CLRBYT C@
  6       40 * OR SCD) CPUT
  7    LOOP
  8   ENDIF ;
  9
 10
 11
 12
 13
 14
 15                                 ==)
```

```
Screen: 109
  0 ( Graphics:  [G"]   G"              )
  1
  2 : (G")                          ( -- )
  3   R COUNT DUP 1+ R) + )R
  4   GTYPE ;
  5
  6 : G"                            ( -- )
  7   22 STATE @
  8   IF
  9     COMPILE (G")
 10     WORD HERE C@ 1+ ALLOT
 11   ELSE
 12     WORD HERE COUNT GTYPE
 13   ENDIF  ; IMMEDIATE
 14
 15                                 --)
```

```
Screen: 110
  0 ( Graphics:  GCOM  DR.  FIL         )
  1
  2 CODE GCOM                     ( n -- )
  3   86 C, D1 C, B5 C, 00 C,
  4   A2 C, 30 C, 9D C, 342 ,
  5   20 C, CIO , A6 C, D1 C,
  6   4C C, POP ,
  7
  8 : DR.                       ( x y -- )
  9   CLRBYT C@ 2FB C!
 10   POS. 11 GCOM ;
 11
 12 : DRAWTO DR. ;
 13
 14 : FIL                   ( fildat -- )
 15   2FD C!  12 GCOM ;   BASE !
```

```
Screen: 111
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 112
  0 ( Graphics Demo                     )
  1 BASE @ DCX
  2
  3 : BOX
  4   1 COLOR 20 10 POSIT
  5   50 10 DR.  50 28 DR.
  6   20 28 DR.  20 10 DR. ;
  7
  8 : FBOX
  9   5 GR. BOX
 10   20 28 POS. 2 FIL ;
 11
 12
 13 ( LOAD THIS SCREEN AND EXECUTE )
 14 ( FBOX.  WHEN YOU'RE DONE, DO  )
 15 ( FORGET BOX )          BASE !
```

```
Screen: 113
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 114
  0 ( Sound:  SOUND  SO.  FILTER!  )
  1
  2 BASE @ HEX
  3 0 VARIABLE AUDCTL
  4
  5 : SOUND  ( ch# freq dist vol --)
  6   3 DUP D20F C! 232 C!
  7   SWAP 10 * + ROT 2*
  8   D200 + ROT OVER C! 1+ C!
  9   AUDCTL C@ D208 C! ;
 10
 11 : SO. SOUND ;
 12
 13 : FILTER!                ( b -- )
 14   DUP D208 C! AUDCTL ! ;
 15                              ==>
```

```
Screen: 115
  0 ( Sound:  XSND  XSND4          )
  1
  2 DCX
  3
  4 : XSND              ( voice# -- )
  5   2* 53761 +
  6   0 SWAP C! ;
  7
  8 : XSND4                    ( -- )
  9   53760 8 0 FILL
 10   0 FILTER! ;
 11
 12
 13
 14
 15                        BASE !
```

```
Screen: 116
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 117
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 118
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 119
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 120
 0 ( Floating:  FDROP FDUP FSWAP  )
 1
 2 BASE @  HEX
 3
 4 CODE FDROP                ( fp --)
 5   INX, INX, POPTWO JMP,
 6 C;
 7
 8 CODE FDUP          ( fp -- fp fp )
 9   # 6 LDY,
10   BEGIN,
11    DEX,  6 ,X LDA,  0 ,X STA,
12    DEY, 0=
13   UNTIL, NEXT JMP,
14 C;
15                                ==>
```

```
Screen: 121
 0 ( Floating:  FSWAP  FOVER       )
 1
 2 CODE FSWAP ( fp1 fp2 -- fp2 fp1)
 3   XSAVE STX,  # 6 LDY,
 4   BEGIN,
 5    0 ,X LDA,  PHA,  6 ,X LDA,
 6    0 ,X STA,  PLA,  6 ,X STA,
 7    INX, DEY, 0=
 8   UNTIL, XSAVE LDX, NEXT JMP, C;
 9
10 CODE FOVER ( fp fp -- fp fp fp )
11   # 6 LDY,
12   BEGIN,
13    DEX,  0C ,X LDA,  0 ,X STA,
14    DEY, 0=
15   UNTIL, NEXT JMP, C;          -->
```

```
Screen: 122
 0 ( Floating:  conversion words  )
 1
 2
 3 CODE AFP
 4   XS, D800 JSR, XL, NXT,
 5 C;
 6
 7
 8 CODE FASC
 9   XS, D8E6 JSR, XL, NXT,
10 C;
11
12
13
14
15                                ==>
```

```
Screen: 123
 0 ( Floating:  FADD FSUB FMUL ...)
 1
 2 CODE IFP  XS, D9AA JSR, XL, NXT,
 3
 4 CODE FPI  XS, D9D2 JSR, XL, NXT,
 5
 6 CODE FADD XS, DA66 JSR, XL, NXT,
 7
 8 CODE FSUB XS, DA60 JSR, XL, NXT,
 9
10 CODE FMUL XS, DADB JSR, XL, NXT,
11
12 CODE FDIV XS, DB28 JSR, XL, NXT,
13
14 CODE FLG  XS, DECD JSR, XL, NXT,
15                                -->
```

```
Screen: 124
 0 ( Floating:  FLG10  FEX  FPOLY )
 1
 2 CODE FLG10
 3   XS, DED1 JSR, XL, NXT,  C;
 4
 5 CODE FEX
 6   XS, DDC0 JSR, XL, NXT,  C;
 7
 8 CODE FEX10
 9   XS, DDCC JSR, XL, NXT,  C;
10
11 CODE FPOLY
12   XS,  0 ,X LDA,  PHA,
13   3 ,X LDA,  XSAVE LDY,
14   2 ,Y LDX,  TAY,  PLA,
15   DD40 JSR,  XL, POP2,  C;   ==>
```

```
Screen: 125
 0 ( Floating:  system constants  )
 1
 2 D4 CONSTANT FR0
 3 E0 CONSTANT FR1
 4 F3 CONSTANT INBUF
 5 F2 CONSTANT CIX
 6
 7
 8
 9
10
11
12
13
14
15                                -->
```

```
Screen: 126
 0 ( Floating:  F@  F!  F.TY       )
 1
 2 : F@                      ( a -- fp )
 3   )R R @ R 2+
 4   @ R) 4 + @ ;
 5
 6 : F!                      ( fp a -- )
 7   )R R 4 + !
 8   R 2+ ! R) ! ;
 9
10 : F.TY                       ( a -- )
11   BEGIN
12     INBUF @ C@ DUP
13     7F AND EMIT
14     1 INBUF +! 80 )
15   UNTIL ;                          ==)
```

```
Screen: 127
 0 ( Floating:  F.  F?  (F  )F     )
 1
 2 : F.                          ( fp -- )
 3   FR0 F@ FSWAP FR0 F! FASC
 4   F.TY SPACE FR0 F! ;
 5
 6 : F?                          ( a -- )
 7   F@ F. ;
 8
 9 : (F                      ( fp fp -- )
10   FR1 F! FR0 F! ;
11
12 : )F                         ( -- fp )
13   FR0 F@ ;
14
15                                    --)
```

```
Screen: 128
 0 ( Floating:  FS  floating +-*/ )
 1
 2 : FS                          ( fp -- )
 3   FR0 F! ;
 4
 5 : F+                      ( fp fp -- fp )
 6   (F FADD )F  ;
 7
 8 : F-                      ( fp fp -- fp )
 9   (F FSUB )F  ;
10
11 : F*                      ( fp fp -- fp )
12   (F FMUL )F  ;
13
14 : F/                      ( fp fp -- fp )
15   (F FDIV )F  ;                    ==)
```

```
Screen: 129
 0 ( Floating: FLOAT FIX FLOG FEXP)
 1
 2 : FLOAT                       ( n -- fp )
 3   FR0 ! IFP )F  ;
 4
 5 : FIX                         ( fp -- n )
 6   FS FPI FR0 @ ;
 7
 8 : LOG                         ( fp -- fp )
 9   FS FLG )F  ;
10
11 : LOG10                      ( fp -- fp )
12   FS FLG10 )F  ;
13
14 : EXP                        ( fp -- fp )
15   FS FEX )F  ;                       --)
```

```
Screen: 130
 0 ( Floating: FEXP10 ASCF FLIT...)
 1
 2 : EXP10                       ( fp -- fp )
 3   FS FEX10 )F  ;
 4
 5 : ASCF                        ( a -- fp )
 6   0 CIX ! INBUF ! AFP )F ;
 7
 8 : FLIT  ( in dict. only: -- fp )
 9   R) DUP 6 + )R F@ ;
10
11 : FLITERAL                ( fp -- [fp])
12   STATE @
13   IF
14    COMPILE FLIT HERE F! 6 ALLOT
15   ENDIF ;   IMMEDIATE             ==)
```

```
Screen: 131
 0 ( Floating:  FLOATING  FP       )
 1
 2 : FLOATING          ( nnnn, -- fp )
 3   BL WORD HERE 1+ ASCF
 4   [COMPILE] FLITERAL ; IMMEDIATE
 5
 6 ( Float the following literal )
 7 ( Ex:  FLOATING 1.2345 )
 8 ( or   FLOATING  -1.67E-13 )
 9
10 : FP                  ( nnnn, -- fp )
11   [COMPILE] FLOATING ;
12   IMMEDIATE
13
14
15                                    --)
```

```
Screen: 132
 0 ( Floating: FVARIABLE FCONSTANT)
 1
 2 : FVARIABLE          ( xxxx, fp -- )
 3                      ( xxxx: -- a  )
 4   <BUILDS
 5     HERE F! 6 ALLOT
 6   DOES> ;
 7
 8 : FCONSTANT          ( xxxx, fp -- )
 9                      ( xxxx: -- fp )
10   <BUILDS
11     HERE F! 6 ALLOT
12   DOES> F@ ;
13
14
15                                  ==>
```

```
Screen: 133
 0 ( Floating:  F0=  F=  F<  F>   )
 1
 2 : F0=                 ( fp -- f )
 3   OR OR 0= ;
 4
 5 : F=                  ( fp fp -- f )
 6   F- F0= ;
 7
 8 : F<                  ( fp fp -- f )
 9   F- DROP DROP 80 AND 0> ;
10
11 : F>                  ( fp fp -- f )
12   FSWAP F< ;
13
14
15                    BASE !
```

```
Screen: 134
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 135
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 136
 0 ( Screen code conversion words )
 1
 2 BASE @ HEX
 3
 4 CODE >BSCD                ( a a n -- )
 5   A9 C, 03 C, 20 C, SETUP ,
 6   HERE  C4 C, C2 C, D0 C, 07 C,
 7   C6 C, C3 C, 10 C, 03 C, 4C C,
 8   NEXT ,       B1 C, C6 C, 48 C,
 9   29 C, 7F C, C9 C, 60 C, B0 C,
10   0D C, C9 C, 20 C, B0 C, 06 C,
11   18 C, 69 C, 40 C, 4C C, HERE
12 2 ALLOT 38 C, E9 C, 20 C, HERE
13 SWAP !  91 C, C4 C, 68 C, 29 C,
14
15                                  ==>
```

```
Screen: 137
 0 ( Screen code conversion words )
 1
 2   80 C, 11 C, C4 C, 91 C, C4 C,
 3   C8 C, D0 C, D3 C, E6 C, C7 C,
 4   E6 C, C5 C, 4C C, ,       C;
 5
 6 CODE BSCD>                ( a a n -- )
 7   A9 C, 03 C, 20 C, SETUP ,
 8   HERE  C4 C, C2 C, D0 C, 07 C,
 9   C6 C, C3 C, 10 C, 03 C, 4C C,
10   NEXT ,       B1 C, C6 C, 48 C,
11   29 C, 7F C, C9 C, 60 C, B0 C,
12   0D C, C9 C, 40 C, B0 C, 06 C,
13   18 C, 69 C, 20 C, 4C C, HERE
14 2 ALLOT 38 C, E9 C, 40 C, HERE
15                                  -->
```

```
Screen: 138
  0 ( Screen code conversion words )
  1
  2 SWAP !  91 C, C4 C, 68 C, 29 C,
  3   80 C, 11 C, C4 C, 91 C, C4 C,
  4   C8 C, D0 C, D3 C, E6 C, C7 C,
  5   E6 C, C5 C, 4C C, ,
  6
  7
  8 : )SCD  SP@ DUP 1 )BSCD ;
  9 : SCD)  SP@ DUP 1 BSCD) ;
 10
 11
 12
 13
 14
 15                        BASE !
```

```
Screen: 139
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 140
  0 ( ValFORTH Video editor   V1.0 )
  1
  2 BASE @ DCX '( )SCD )( 68 KLOAD )
  3
  4 VOCABULARY EDITOR IMMEDIATE
  5 EDITOR DEFINITIONS
  6
  7 0 VARIABLE XLOC    ( X coord.    )
  8 0 VARIABLE YLOC    ( Y coord.    )
  9 0 VARIABLE LSTCHR ( last key    )
 10 0 VARIABLE ?ESC    ( coded char?)
 11 0 VARIABLE TBLK    ( top block   )
 12 0 VARIABLE UPSTAT 2 ALLOT ( map)
 13
 14  15 CONSTANT 15  32 CONSTANT 32
 15 128 CONSTANT 128              ==)
```

```
Screen: 141
  0 ( ValFORTH Video editor   V1.0 )
  1
  2 : LMOVE 32 CMOVE ;
  3
  4 : BOL 88 @ YLOC @ 1+ 32 * + ;
  5
  6 : SBL  88 @ 544 + ;
  7
  8 : CURLOC                  ( --- )
  9   BOL XLOC @ + ;     ( SCR ADDR )
 10
 11 : CSHOW                   ( --- )
 12   CURLOC DUP       ( GET SCR ADDR )
 13   C@ 128 OR        ( INVERSE CHAR )
 14   SWAP C! ;        ( STORE ON SCR )
 15                                --)
```

```
Screen: 142
  0 ( ValFORTH Video editor   V1.0 )
  1
  2 : CBLANK                  ( --- )
  3   CURLOC DUP C@ 127
  4   AND SWAP C! ;
  5
  6 : UPCUR                    ( -- )
  7   CBLANK YLOC @ 1- DUP
  8   0( IF DROP 15 ENDIF
  9   YLOC ! CSHOW ;
 10
 11 : DNCUR                    ( -- )
 12   CBLANK YLOC @
 13   1 + DUP 15 )
 14   IF DROP 0 ENDIF
 15   YLOC ! CSHOW ;               ==)
```

```
Screen: 143
  0 ( ValFORTH Video editor   V1.0 )
  1
  2 : LFCUR                    ( -- )
  3   CBLANK XLOC @
  4   1 - DUP 0(          ( AT L-SIDE?)
  5   IF DROP 31 ENDIF  ( FIX IF SO )
  6   XLOC ! CSHOW ;
  7
  8 : RTCUR                    ( -- )
  9   CBLANK XLOC @
 10   1+ DUP 31 )         ( AT R-SIDE?)
 11   IF DROP 0 ENDIF   ( FIX IF SO )
 12   XLOC ! CSHOW ;
 13
 14 : EDMRK
 15   1 YLOC @ 4 / UPSTAT + C! ; --)
```

```
Screen: 144
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : LNINS                        ( -- )
  3   CBLANK
  4   4 YLOC @ 4 /
  5   DO 1 I UPSTAT + C! LOOP
  6   YLOC @ 15 <
  7   IF
  8     BOL DUP 32 +
  9     15 YLOC @ - 32 *
 10     <CMOVE
 11   ENDIF
 12   BOL 32 ERASE
 13   CSHOW EDMRK ;
 14
 15                                ==>
```

```
Screen: 145
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : LNDEL                        ( -- )
  3   CBLANK
  4   4 YLOC @ 4 /
  5   DO 1 I UPSTAT + C! LOOP
  6   YLOC @ 15 <
  7   IF BOL                   ( FROM )
  8     DUP 32 + SWAP           ( TO   )
  9     15 YLOC @ - 32 *        ( # CH )
 10     CMOVE
 11   ENDIF
 12   BOL 15 YLOC @ -
 13   32 * + 32 ERASE
 14   CSHOW EDMRK ;
 15                                -->
```

```
Screen: 146
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : RUB                          ( -- )
  3   XLOC @ 0= NOT    ( ON L-EDGE? )
  4   IF LFCUR 0 CURLOC C!
  5      CSHOW EDMRK
  6   ENDIF ;
  7
  8 : PTCHR                        ( -- )
  9   EDMRK
 10   LSTCHR @ 127 AND
 11   DUP LSTCHR !
 12   >SCD CURLOC C!
 13   RTCUR XLOC @ 0=
 14   IF DNCUR ENDIF
 15   0 ?ESC ! CSHOW ;             ==>
```

```
Screen: 147
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : SCRSV                        ( -- )
  3   88 @ 32 + PAD 512 BSCD>
  4   4 0
  5   DO
  6     I UPSTAT + C@
  7     0 I UPSTAT + C!
  8     IF
  9       PAD 128 I * +
 10       TBLK @ I + BLOCK
 11       128 CMOVE UPDATE
 12     ENDIF
 13   LOOP
 14   0 XLOC ! 0 YLOC ! ;
 15                                -->
```

```
Screen: 148
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : SCRGT                        ( -- )
  3   4 0
  4   DO
  5     TBLK @
  6     I + BLOCK
  7     PAD 128 I * +
  8     128 CMOVE
  9   LOOP
 10   PAD 88 @ 32 +
 11   512 >BSCD ;
 12
 13
 14
 15                                ==>
```

```
Screen: 149
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : NWSCR              ( -1/0/1 -- )
  3   CBLANK DUP
  4   IF SCRSV ENDIF 2* 2*
  5   TBLK @ + 0 MAX TBLK ! SCRGT
  6   TBLK @ 8 /MOD
  7   DUP <ROT SCR !
  8   IF 44 ELSE 53 ENDIF
  9   ?1K NOT
 10   IF
 11     44 = SWAP 2* + DUP SCR ! 0
 12   ENDIF
 13   88 @ 17 + C!
 14   0 84 C! 11 85 ! 1 752 C!
 15   . 2 SPACES CSHOW ;           -->
```

```
Screen: 150
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : SPLCHR  1 ?ESC ! ;          ( -- )
  3
  4 : EXIT                        ( -- )
  5   CBLANK 19 LSTCHR !
  6   0 XLOC ! 0 YLOC ! ;
  7
  8 : EDTABT                      ( -- )
  9   UPSTAT 4 0 FILL
 10   EXIT ;
 11
 12
 13
 14
 15                                  ==>
```

```
Screen: 151
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : CONTROL                   ( n -- )
  3   DUP  19 = IF DROP EXIT    ELSE
  4   DUP  17 = IF DROP EDTABT  ELSE
  5   DUP  28 = IF DROP UPCUR   ELSE
  6   DUP  29 = IF DROP DNCUR   ELSE
  7   DUP  30 = IF DROP LFCUR   ELSE
  8   DUP  31 = IF DROP RTCUR   ELSE
  9   DUP 126 = IF DROP RUB     ELSE
 10   DUP 157 = IF DROP LNINS   ELSE
 11   DUP 156 = IF DROP LNDEL   ELSE
 12        27 = IF DROP SPLCHR  ELSE
 13
 14
 15                                  -->
```

```
Screen: 152
  0 ( ValFORTH Video editor    V1.0 )
  1
  2   PTCHR    ( IF NOTHING SPECIAL )
  3   ENDIF ENDIF ENDIF ENDIF
  4   ENDIF ENDIF ENDIF ENDIF
  5   ENDIF ENDIF ;
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15                                  ==>
```

```
Screen: 153
  0 ( ValFORTH Video editor    V1.0 )
  1
  2 : (V)                     ( TBLK -- )
  3   DECIMAL
  4   DUP BLOCK DROP TBLK !
  5   UPSTAT 4 0 FILL
  6   1 PFLAG ! 0 GR.
  7   1 752 C! CLS
  8   1 559 C@ 252
  9   AND OR 559 C!
 10   112 560 @ 6 + C!
 11   112 560 @ 23 + C!
 12   ." Screen #" 11 SPACES
 13   ." ValFORTH"
 14   0 NWSCR
 15                                  -->
```

```
Screen: 154
  0 ( ValFORTH Video editor    V1.0 )
  1
  2
  3 ( Main loop of editor )
  4
  5   BEGIN
  6     KEY DUP LSTCHR !
  7     ?ESC @
  8     IF
  9       PTCHR 0 LSTCHR !
 10     ELSE
 11       CONTROL
 12     ENDIF
 13     LSTCHR @ 19 =
 14   UNTIL
 15                                  ==>
```

```
Screen: 155
  0 ( ValFORTH Video editor    V1.0 )
  1
  2   CBLANK SCRSV 0 767 C!
  3   2 560 @ 6 + C!
  4   2 560 @ 23 + C!
  5   2 559 C@ 252
  6   AND OR 559 C!
  7   0 752 C! CLS CR
  8   ." Last edit on screen # "
  9   SCR @ . CR CR ;
 10
 11 FORTH DEFINITIONS
 12
 13 : V                          ( s -- )
 14   1 MAX B/SCR *
 15   EDITOR (V) ;                   -->
```

```
Screen: 156
  0 ( ValFORTH Video editor   V1.0 )
  1
  2 : L                          ( -- )
  3   SCR @ DUP 1+
  4   B/SCR * SWAP B/SCR *
  5   EDITOR TBLK @ DUP <ROT
  6   <= <ROT > AND
  7   IF
  8     EDITOR TBLK @
  9   ELSE
 10     SCR @ B/SCR *
 11   ENDIF
 12   EDITOR (V) ;
 13
 14
 15                                ==>
```

```
Screen: 157
  0 ( ValFORTH Video editor   V1.0 )
  1
  2 : CLEAR                    ( s -- )
  3   B/SCR * B/SCR 0+S
  4   DO
  5     FORTH I BLOCK
  6     B/BUF BLANKS UPDATE
  7   LOOP ;
  8
  9 : COPY                ( s1 s2 -- )
 10   B/SCR * OFFSET @ +
 11   SWAP B/SCR * B/SCR 0+S
 12   DO DUP FORTH I
 13     BLOCK 2- !
 14     1+ UPDATE
 15   LOOP DROP ;                  -->
```

```
Screen: 158
  0 ( ValFORTH Video editor   V1.0 )
  1
  2
  3 ( Note: the fig bug is fixed )
  4 (       in WHERE below.      )
  5
  6 HEX
  7 : WHERE              ( [ n n ] -- )
  8   2DUP DUP B/SCR / DUP SCR !
  9   ." Scr # " DECIMAL . SWAP
 10   C/L /MOD C/L * ROT BLOCK +
 11   CR C/L -TRAILING TYPE
 12   CR HERE C@ - 2- 0 MAX SPACES
 13   1 2FE C! 1C EMIT 0 2FE C!
 14   [COMPILE] EDITOR QUIT ;
 15                      BASE !
```

```
Screen: 159
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 160
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 161
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 162
  0 ( DOS:  input/output routines  )
  1
  2 BASE @ HEX
  3
  4 340 VARIABLE IOCB
  5   0 VARIABLE IO.X
  6   0 VARIABLE IO.CH
  7
  8 : IOCC
  9   10 * 70 MIN DUP IO.X C!
 10   340 + IOCB !  ;
 11
 12 : (IO)
 13   (BUILDS ,
 14   DOES) @ IOCB @ + ;
 15                              ==)
```

```
Screen: 163
  0 ( DOS:  system words           )
  1
  2 2 (IO) ICCOM  3 (IO) ICSTA
  3 4 (IO) ICBAL  8 (IO) ICBLL
  4 A (IO) ICAX1  B (IO) ICAX2
  5 C (IO) ICAX3  D (IO) ICAX4
  6 E (IO) ICAX5  F (IO) ICAX6
  7
  8
  9 CODE XCIO
 10   XSAVE STX, IO.X LDX,
 11   IO.CH LDA, E456 JSR,
 12   XSAVE LDX, IO.CH STA,
 13   TYA, PUSH0A JMP,
 14 C;
 15                              -->
```

```
Screen: 164
  0 ( DOS:  OPEN CLOSE PUTC GETC   )
  1
  2 : OPEN    ( adr n1 n2 n3 -- n4 )
  3   IOCC ICAX2 C! ICAX1 C!
  4   ICBAL ! 03 ICCOM C! XCIO ;
  5
  6 : CLOSE                 ( n1 -- )
  7   IOCC 0C ICCOM C! XCIO DROP ;
  8
  9 : PUT               ( c n1 -- n2 )
 10   IOCC IO.CH C! 0B
 11   ICCOM C! XCIO ;
 12
 13 : GET               ( n1 -- c n2 )
 14   IOCC 7 ICCOM C! XCIO
 15   IO.CH C@ SWAP ;             ==)
```

```
Screen: 165
  0 ( DOS:  GET/PUTREC STATUS DEV  )
  1
  2 : GETREC     ( adr n1 n2 -- n3 )
  3   IOCC 5 ICCOM C! ICBLL !
  4   ICBAL ! XCIO ;
  5
  6 : PUTREC     ( adr n1 n2 -- n3 )
  7   IOCC 9 ICCOM C! ICBLL !
  8   ICBAL ! XCIO ;
  9
 10 : STATUS            ( n1 -- n2 )
 11   IOCC ICSTA C@ ;
 12
 13 : DEVSTAT     ( n1 -- n2 n3 n4 )
 14   IOCC 0D ICCOM C! XCIO
 15   )R 2EA @ 2EC @ R) ;         -->
```

```
Screen: 166
  0 ( DOS:  SPECIAL                )
  1
  2 : SPECIAL
  3 ( n1 n2 n3 n4 n5 n6 n7 n8 -- n9)
  4   IOCC ICCOM C! ICAX6 C!
  5   ICAX5 C! ICAX4 C! ICAX3 C!
  6   ICAX2 C! ICAX1 C! XCIO ;
  7
  8
  9
 10
 11
 12
 13
 14
 15                          BASE !
```

```
Screen: 167
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen: 168
 0 ( Atari 850 download                  )
 1
 2 BASE @ HEX
 3
 4 CODE DO-SIO
 5   XSAVE STX, 0 # LDA,
 6   E459 JSR,
 7   XSAVE LDX, NEXT JMP,
 8
 9 : SET-DCB
10    50 300 C!     1 301 C!
11    3F 302 C!    40 303 C!
12   500 304  !     5 306 C!
13     0 307 C!     C 308 C!
14     0 309  !     0 30B C! ;
15                                  ==>
```

```
Screen: 169
 0 ( Atari 850 download                  )
 1
 2 CODE RELOCATE
 3   XSAVE STX, 506 JSR,
 4   HERE 8 + JSR,  XSAVE LDX,
 5   NEXT JMP,   0C )JMP,
 6
 7
 8 : RS232                        ( -- )
 9   HERE 2E7 ! SET-DCB DO-SIO
10   500 300 0C CMOVE DO-SIO
11   RELOCATE 2E7 @ HERE - ALLOT
12   HERE FENCE ! ;
13
14
15                        BASE !
```

```
Screen: 170
 0 CONTENTS OF THIS DISK:
 1
 2 PRINTER UTILITIES:       38 LOAD
 3 DEBUGGING AIDS:          42 LOAD
 4 VALFORTH EDITOR 1.0:    140 LOAD
 5 ASSEMBLER:               76 LOAD
 6 COLOR COMMANDS:         100 LOAD
 7 GRAPHICS:               104 LOAD
 8 GRAPHICS DEMO:          112 LOAD
 9 SOUNDS:                 114 LOAD
10 FLOATING POINT:         120 LOAD
11  (FP REQUIRES ASSEMBLER FIRST)
12 SCREEN CODE CONVERS.:   136 LOAD
13 FORMATTER:               92 LOAD
14 DISK COPIERS:            72 LOAD
15   (continued on next screen)
```

```
Screen: 171
 0 CONTENTS OF THIS DISK, cont:
 1
 2 fig EDITOR:              56 LOAD
 3 BUFFER RELOCATION:       94 LOAD
 4 AUTO-BOOT UTILITY:       30 LOAD
 5 OPERATING SYS. WORDS:   162 LOAD
 6 850 DOWNLOAD (RS-232): 168 LOAD
 7  (OPSYS AND 850 NEED ASSEMBLER)
 8
 9
10
11
12
13
14
15
```

```
Screen: 172
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 173
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 174
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 175
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 176
 0 ( Error messages                )
 1
 2 Stack empty
 3
 4 Dictionary full
 5
 6 Wrong addressing mode
 7
 8 Is not unique
 9
10 Value error
11
12 Disk address error
13
14 Stack full
15
```

```
Screen: 177
 0 Disk Error!
 1
 2 Dictionary too big
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Screen: 178
 0 ( Error messages                )
 1
 2 Use only in Definitions
 3
 4 Execution only
 5
 6 Conditionals not paired
 7
 8 Definition not finished
 9
10 In protected dictionary
11
12 Use only when loading
13
14 Off current screen
15
```

```
Screen: 179
 0 Declare VOCABULARY
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```